New Production System for Finnish Meteorological Institute

Santeri Horttanainen

Helsinki January 16, 2019

UNIVERSITY OF HELSINKI
Department of Computer Science

HELSINGIN YLIOPISTO — HELSINGFORS UNIVERSITET — UNIVERSITY OF HELSINKI

Tiedekunta — Fakultet — Faculty	Laitos — Institution — Department
Faculty of Science	Department of Computer Science

Tekijä — Författare — Author
Santeri Horttanainen

Työn nimi — Arbetets titel — Title

New Production System for Finnish Meteorological Institute

Oppiaine — Läroämne — Subject
Computer Science

Työn laji — Arbetets art — Level	Aika — Datum — Month and year	Sivumäärä — Sidoantal — Number of pages
	January 16, 2019	85 pages + 2 appendices

Tiivistelmä — Referat — Abstract

This thesis presents the plans for replacing the production system of Finnish Meteorological Institute (FMI). It begins with a review of the state of the art in distributed systems research, and ends with a design for the replacement production system that is reliable, scalable, and maintainable.

The subject production system is a framework for managing the production of different weather predictions and models. We use this framework to abstract away the actual execution of work from its description. This way the different production processes become easily monitored and configured through the production system.

Since the amount of data processed by this system is too much for a single computer to handle, we have distributed the production system. Thus we are not dealing with just a framework for production but with a distributed system and hence a solid understanding of distributed systems theory is required in order to replace this production system.

The first part of this thesis lays the groundwork for replacing the distributed production system: a review of the state of the art in distributed systems research. It is a concise document of its own which presents the essentials of distributed systems in a clear manner. This part can be used separately from the rest of this thesis as a short introduction to distributed systems.

Second part of this thesis presents the subject production system, the need for its replacement, and our design for the new production system that is maintainable, performant, available, reliable, and scalable. We go even further than simply giving a design for this replacement production system, and instead present a practical plan to implement the new production system with Kubernetes, Brigade, and Riak CS.

ACM Computing Classification System (CCS):
Computer systems organization → Distributed Architectures → Cloud computing
Software and its engineering → Software organization and properties → Software system structures → Distributed systems organizing principles → Cloud computing

Avainsanat — Nyckelord — Keywords
Distributed Systems, Production Systems, Software Architecture, Software Engineering

Säilytyspaikka — Förvaringsställe — Where deposited

Muita tietoja — övriga uppgifter — Additional information

Contents

1 Introduction

The main subject of this thesis is the replacement of a legacy production system used at FMI. This system processes raw telemetry data, weather predictions, and weather models and turns them into other weather predictions and models. Conceptually the system under consideration is very simple: data comes in, data comes out. Nevertheless, the implementation is far from simple because of the amount of data processed each day. A single machine would simply be insufficient. This is why the production system is actually a network of hundreds of computers working in unison. To replace this system a solid understanding on distributed systems is first required.

The current production system is problematic in several ways. Load is balanced manually, there is no replication, and the input/output (I/O) of the central Network File System (NFS) cluster is already becoming a bottleneck. The lack of automatic load balancing makes the operability of the system difficult. Whenever a task is added to the system the operator has to make the difficult decision of choosing the worker it is assigned to. Moreover, this often causes the system to become unbalanced. The lack of replication leads to the system being very vulnerable: if a component fails we lose all its functionality. The unavailability of a single worker means that its tasks will not be executed until that worker becomes available again. The limited I/O of the NFS cluster is a problem by itself, but in the distributed systems sense this leads to an another problem: since all the data is accessed through a central cluster, the scalability of the system is limited by this bottleneck.

But there is an even more pressing need to update the current production system: The open data movement is catching on, meaning that new interesting data is made available around the world. But currently we are only able to access a subset of this data: data that is either sufficiently small or updated not too frequently. The reason is that currently all data to be processed by the current system needs first to be downloaded to the central NFS cluster. This means that we are restricted by the size and update frequency of the data: there is simply no point in processing data that is stale by the time its download is done. We need to change the current solution so that tasks can be sent to the data rather than the other way around.

The main contributions of this thesis are the design for the replacement production system and the preliminary research on distributed systems theory, which may be used as a short introduction to distributed systems. The design for the new produc-

tion system counters the shortcomings of the current one and is designed to be as simple as possible, to ensure maintainability in the future. To accompany the design of the new production system, we compared several off-the-shelf software products to find the most suitable ones for actual implementation of the new design.

The remainder of this thesis is organised as follows: Section 2 presents background information on production systems. Section 3 presents background information on distributed systems. Section 4 reviews the state of the art in distributed systems. Section 8 outlines the problem with the current production system. Section 9 presents the design for the replacement production system and describes how the design can be implemented with off-the-shelf software. Finally, section 10 provides conclusions and suggestions for future work.

2 A Brief Review of Production Systems

The term production system is unfortunately fairly ambiguous. Specifically, the subject production system of this thesis should not be confused with production rule system (sometimes shortened to just production system), a branch of artificial intelligence [1]. The production system under consideration is more like the Toyota Production System (TPS) [2], but instead of cars the production system produces different kinds of data such as weather forecast models, animations, and pictures.

Definition 1. Production system is a system that controls production processes [3, Chapter 2]. Specifically, a production system controls resources such as humans, premises, machines, and equipment to transform input materials into desired output products [3, Chapter 2].

Informally, a production system is a framework for managing production. The idea behind production system is to require operators to make their processes of production fit into the framework. This results in differing workflows becoming more similar to each other, which helps in their management and monitoring. Consequently, using a production system enables operators to quickly respond to changes: operators no longer need to change each production process individually, since all production processes are fit to the framework, operators can quickly change the direction of the whole production by changing the configurations of the framework. Similarly production system also enables operators to easily monitor the performance of different workflows.

3 A Brief Review of Distributed Systems

This section gives background information on distributed systems for readers not already familiar with the subject. First, subsection 3.1 presents a definition for a distributed system. Next, subsection 3.2 describes a set of goals we should aim for when designing a distributed system. Finally, subsection 3.3 explains why distributed systems are difficult to implement.

3.1 Definition

The word *distributed* in *distributed systems* comes from geographic distribution [4, Chapter 1]. This geographic distribution might mean anything from computers that are separated by only a few centimeters (e.g. servers placed in a rack and connected via a local area network (LAN)) to thousands of kilometers (e.g. data centres forming an overlay network over a wide area network (WAN)).

Definition 2. Distributed system is a collection of independent networked computers appearing to its users as a single coherent system [4, Chapter 1].

Distributed systems should not be grouped together with parallel computing systems of high performance computing (HPC). In parallel computing systems nodes are central processing unit (CPU) cores whereas in distributed systems nodes are complete computers with their own random-access memory (RAM), disks, and multi-core CPUs. This makes the computing units (nodes) of distributed systems a lot more autonomous than the computing units (CPUs) of parallel computing systems. Another aspect where parallel computing systems greatly differ from distributed systems is how much more closely connected the nodes are in parallel computing systems. Instead of being connected via a LAN (or via a WAN) nodes in parallel computing systems are connected via a bus. Moreover, nodes in parallel computing systems frequently share RAM and a master clock, making synchronised computation easy and fast. Contrastingly, nodes in distributed systems rarely share anything besides the network connecting them.

3.2 Designing for Reliability, Scalability, and Maintainability

Before designing a distributed system, we need a clear set of goals to aim for. Since the matter of distributed systems is a complicated subject, it is best to keep these

goals simple. When designing a distributed system, we are looking for reliability, scalability, and maintainability. Rest of this thesis is about fulfilling these goals.

3.2.1 Reliability

We want distributed systems to be reliable: to tolerate hardware and software faults as well as human errors to the extent that is possible. Understanding the difference between fault and failure is key to understanding reliability [5]. A system is said to have encountered a failure if it deviates from its specification for a period of time [5]. A fault, on the other hand, is the cause of a failure [5]: a fault of a component, a subsystem, or a another system interacting with the considered (failed) system. If a system with faults can continue to provide its service, its said to be fault-tolerant [5]. The more fault-tolerant a system is, the more reliable it is.

3.2.2 Scalability

We want distributed systems to be scalable: to be able to cope with increased load. A scalable system handles the addition of users, data, or geographical distance without an obvious loss in performance [6].

The better a system scales with size, the more linearly its performance increases with number of nodes [7, Chapter 1]. For example, a system of n nodes that needs $\mathcal{O}(n^2)$ messages for every decision does not scale well with its size since a small increase of one node $(n+1)$ would translate into an exponential increase $((n+1)^2)$ in messages required for reaching a decision.

Geographical scalability is the ability of the system to handle gracefully the growing latencies caused by the increasing distances between the farthest nodes in the system [6]. Geographical scaling is important to acknowledge because reasonable approaches in a high-speed, low-latency LAN might perform poorly in WAN.

3.2.3 Maintainability

Majority of the cost of software is not in its initial deployment, but in its ongoing maintenance [7, Chapter 1]. This is doubly more so for distributed systems, which are complex by nature and hence keeping them maintainable is of utmost importance. Good maintainability can be achieved by two design principles: operability and simplicity.

Operability measures how easy it is for the operators to run the system smoothly (and to know when it is not doing so). Techniques to achieve good operability include: providing good visibility into the internals of the system (good monitoring, logging, and debugging equipment) [7, Chapter 1], enabling rolling updates (e.g. through fault-tolerance at the node-level) [8, Chapter 7], and providing good integration with standard tools [7, Chapter 1].

Simplicity of system indicates how easy it is for new operators to understand it [7]. The most effective way to achieving simplicity is to emphasise good design decisions in the system architecture and to continuously look for handy abstractions to hide away the complex implementation details behind a simple-to-understand facade [7, Chapter 1].

3.3 Difficulties

Possibility of partial failures is the defining characteristic of distributed systems. Any operation involving multiple nodes might or might not work, or it might even work on only some subset of the nodes, leading to an inconsistent system. Detecting these faults is hard since the network connecting the nodes may occasionally drop, reorder, and arbitrarily delay messages. Consequently, what might look like a failure, might equally well be a complete success, or something first appearing to be a success might later be revealed to have actually failed. Therefore, it is very hard for the nodes in distributed systems to agree on anything, not even on the time.

3.3.1 Partial Failures

Distributed systems fail frequently and often partially [9]. An individual computer with well written software is usually either fully functional or entirely broken [7, Chapter 8]. Unfortunately, this is not the case for distributed systems: there may well be some parts in the system that are broken, even though other parts of the system are perfectly fine. Network switches fail, garbage collector (GC) pauses cause leaders to disappear, failing components cause nodes to become unavailable, some writes seem to succeed but actually fail at the receiving end, and individual staggers cause whole clusters to crawl [9]. What makes partial failures difficult to handle, is that they are non-deterministic: any operation involving multiple nodes may work fine most of the time, but sometimes suddenly fail [7, Chapter 8]. Such failures are so common for systems of scale that dealing with them should be considered as

standard mode of operation [10, 11].

3.3.2 Unreliable Networks

Networks are unreliable [12, 13]. There are plenty of things that can cause networks to fail and in surprising ways [7, Chapter 8]. Sometimes sharks damage undersea network cables by biting them [14]. Sometimes an update for a network switch causes the whole network topology to change in a way that causes packets to stall for more than a minute [15]. Sometimes a network interface just starts dropping all inbound packets while outbound traffic remains unaffected [16]. And sometimes a maintenance misconfiguration causes a whole data centre to become unavailable [17]. But most often the reason is a failing network switch.

Networks are asynchronous. They make no guarantees about when a message will arrive or whether it will arrive at all. Thus, it is impossible to know if a message was received without receiving an acknowledgement, and if acknowledgement is never received, it is impossible to know why [7, Chapter 8]. Missing acknowledgement may mean any of the following things: the message got lost, the message is still on its way, the message was delivered but the recipient failed to reply with an acknowledgement, the message was received but the acknowledgement got lost, or the acknowledgement is still on its way.

This uncertainty of the network makes failure detection difficult. In a system where messages and their acknowledgements may get lost, the only way to detect a failure is by timeout [7, Chapter 8]. But since messages may also be arbitrarily delayed, choosing the correct value for this timeout is difficult. A long timeout increases the certainty of correctly detecting a fault. But the longer we set the timeout, the slower the system becomes at detecting failures. A short timeout, on the other hand, makes the system faster at reacting to failures but with the increased risk of declaring delayed messages as failures.

3.3.3 Unreliable Clocks

Each node has its own clock, usually a quartz crystal oscillator, but unfortunately such hardware devices are never perfectly accurate [7, Chapter 8]. They always run slightly faster or slower than they should (i.e. they drift). Consequently, each node has its own notion of time.

Since clocks drift, they need to be frequently synchronised. But unfortunately the

available synchronisation methods are not very accurate either. Their synchronisation accuracy is limited by the uncertainty of network latency. For example, one study showed a typical root mean square (RMS) value of 35 ms synchronisation error for Network Time Protocol (NTP) (a very common clock synchronisation mechanism) when synchronising over internet [18].

Not only are hardware clocks impossible to keep synchronised, but they might sometimes go backward or suddenly jump forward. This happens when a clock is forcibly reset because of drifting too much apart from the clock of its reference NTP cluster [7, Chapter 8].

4 Fundamental Properties of Distributed Systems

In this section we present the fundamental properties of distributed systems as is relevant to the scope of this thesis. We start by going over different replication strategies in subsection 4.1. Next, we move to discuss ways to detect and to handle conflicts in subsection 4.2. Then, we discuss different sharding strategies and how they complement replication in subsection 4.3. Next, we discuss the differences between various consistency models in subsection 4.4. Lastly, we finish by discussing fault-tolerant consensus and its applications in subsection 4.5.

4.1 Replication

Replication is the act of copying a dataset, or a shard of a dataset, across multiple nodes. These copies are called replicas, which are usually divided into two groups: leaders and followers (except for leaderless replication which does not make this distinction). Leaders differ from followers in that they can accept writes whereas followers can only accept reads.

Replication improves availability by allowing a system to continue to work even if parts of it have failed. Specifically, if a shard is replicated to n nodes it becomes unavailable only after every one of these n nodes has failed. Additionally, replication improves performance by enabling more nodes to handle requests made to a shard. Lastly, replication reduces latency by enabling a replica to be placed geographically close to users located far away.

4.1.1 Asynchronous Versus Synchronous Replication

Whenever a leader accepts a write, the write must be eventually replicated to all followers in order for the system to remain consistent. And sooner or later, the leader has to acknowledge the write to satisfy its origin. The sooner the acknowledgement arrives, the sooner the waiting process can continue with its other responsibilities. This is where the decision between asynchronous and synchronous replication comes to play. Asynchronous replication means acknowledging a write immediately, before waiting for the replication to be done, whereas in synchronous replication the write is only acknowledged after it has been replicated to all synchronous followers.

The advantage of synchronous replication is that the follower is kept consistent with the leader: if the leader fails there remains an up-to-date replica which can then

be promoted to be the new leader [7, Chapter 5]. But synchronous replication also has its downsides. For example, if the follower becomes unavailable, the leader is forced to wait until it becomes available again preventing the leader from accepting new writes [7, Chapter 5]. In addition, the additional replication delay increases the latency of the system.

The advantage of asynchronous replication is faster write response times. Since replication is done in the background, the leader is able to respond immediately. Moreover, asynchronous replication enables leader to continue to accept writes even if all its followers become unavailable. But asynchronous replication also has a downside: if the leader crashes after accepting a couple of writes, while its followers have been unavailable during these writes, then there is a real chance that these writes become lost forever.

4.1.2 Single-Leader Replication

In single-leader replication there is only one leader per shard, which makes this replication strategy conflict-free: without competing leaders there can be no conflicts (see subsection 4.2). Consequently, single-leader replication often leads to considerably simpler design than multi-leader or leaderless replication since it does not require complex conflict handling strategies.

The major disadvantages of single-leader replication are its relatively poor write availability and performance when compared with other replication strategies. Since there is only one leader per shard, its write availability is completely dependent on its leaders availability: if leader becomes unavailable for any reason it will render the whole shard unavailable for writes [7, Chapter 5]. Additionally, because only one node can hold the leader of a replica, the throughput at which the shard can accept writes is limited to the capacity of that particular node. However, this problem can often be alleviated (or completely mitigated) with a sharding strategy that properly captures the communication pattern of the service (see subsection 4.3). As a rule of thumb: the write performance of single-leader replication should not be a problem if writes are mostly consecutive per shard [19].

4.1.3 Multi-Leader Replication

Multi-leader replication provides better write performance and availability than single-leader replication. In multi-leader replication writes can be concurrently ac-

cepted by as many nodes as there are leaders, which can significantly increase the write throughput of the system when compared with single-leader replication where writes are consecutively applied by only one node at a time. Moreover, having multiple leaders improves write availability: a shard becomes unavailable for writes only after all of its leaders have become unavailable, whereas with single-leader replication only one leader failure is too much.

It usually does not make sense to employ multi-leader replication in intra-data centre services because its complexity often outweighs the added benefits [7, Chapter 5]. But sometimes multi-leader replication is required, like when offline operation is required or when a service spans multiple data centres.

Multi-leader replication provides better write performance than single-leader replication in inter-data centre services. For example, if there is only one leader per shard in an inter-data centre service, then the leader of each shard can reside in only one of the data centres at a time. Consequently, all writes originating from other data centres have to be forwarded to the data centre holding the leader, significantly increasing the write latency. This may possibly even nullify the benefits of having multiple data centres in the first place [7, Chapter 5]. In contrast, with a multi-leader system we could assign a leader to each data centre enabling them to accept writes independently of each other.

Multi-leader replication generally provides better write availability than single-leader replication for inter-data centre services. For example, consider a network partition that temporarily prevents communication between some data centres. In single-leader systems only the data centre holding the leader could continue to accept writes, but clients of other data centres would experience write unavailability. Contrastingly, systems with multi-leader replication could continue to accept writes regardless of such partitions.

Multi-leader replication is also better suitable for offline operation than single-leader replication. Mobile devices are constantly going to and coming from offline because of geographic variations in network availability and to save power. In such systems, replication is done only between online replicas: every time a replica becomes online it synchronises itself with other online replicas. Such offline operation is not possible with single-leader replication since the follower replicas would be forced to wait for the leader to become online before accepting any writes [20]. In fact, all devices capable of offline operation have to be leaders as well in order to be able to accept writes autonomously.

But multi-leader replication has one great downside. Having multiple leaders means the ability to accept writes concurrently, but sometimes these writes conflict. Consequently, multi-leader systems require mechanisms to detect and to resolve occasional write conflicts. Unfortunately, this adds a lot of complexity (see subsection 4.2).

4.1.4 Leaderless Replication

In the replication strategies considered so far, only leaders have been allowed to accept writes, but what happens when there are no leaders? Replication without leaders is called leaderless replication [7, Chapter 5]. Unlike in multi-leader replication, where only leaders are allowed to accept writes, in leaderless replication every replica is allowed to accept writes. Although the preceding description might sound like multi-leader replication with every node designated as a leader, there are important differences. For starters, in multi-leader replication leaders can accept requests independently of each other, but in leaderless replication a quorum of replicas is always required to participate. In the following paragraphs we will discuss these differences in more detail.

Systems using leaderless replication are said to be quorum systems: in order for a request to succeed, a preconfigured amount of replicas (the quorum) is required to vote in favour of accepting the request [10]. This vote is usually done by forwarding each request to a shard to all replicas of that shard. In Dynamo, Amazon's distributed key-value store built on leaderless replication, this forwarding is done through a designated coordinator replica [10]: the client sends their request to some node in their Dynamo instance, the receiving node then forwards the request to the coordinator replica of the related partition, the coordinator sends the request to all remaining replicas, and finally decides on the result according to content and number of received replies.

So a request is rejected if it can not reach enough replicas, but how does the system tolerate writes that reach just enough replicas? Would not a single unavailable replica lead to a inconsistent replica state? Dynamo solves this problem by employing a technique called read repair: a minority of replicas can diverge during writes, due to temporary failures, but get reconciled during reads (see subsection 4.2).

Because of the combination of quorum rules with read repair, leaderless replication can tolerate some unavailable replicas and thus be highly available. But exactly how many unavailable replicas can a leaderless replica set tolerate? The answer lies in

the balance between the overall number of replicas N, the number replicas required to participate in writes W, and the number of replicas required to participate in reads R. Now, if we manage to arrange votes so that two consecutive successful votes always have one replica in common, it is guaranteed that all inconsistencies are eventually discovered during read repair, since at least one of the nodes is always guaranteed to be up-to-date. Thus in order to guarantee successful read repairs, we have to set R and W such that $R + W > N$ [7, Chapter 5]. In contrast, setting R and W to smaller values, such that $R + W \leq N$, can lead to failed read repair (a stale value is returned). Systems with low latency requirements can benefit from the latter setting since the lowered quorum requirements directly translate to less replicas having to be waited upon. A good starting configuration of (N,R,W) for many applications is (3,2,2) [10].

A system that requires at least W and R replicas to participate during writes and reads is said to employ strict quorum, but there is yet another (more available) technique called sloppy quorum. Therefore a system using strict quorum can tolerate the unavailability of $N - W$ or $N - R$ replicas depending on the request. Employing strict quorum also grants the system the ability to tolerate individual stragglers since it makes returning possible as soon as R or W replicas have responded. But using strict quorums fails in situations where a large number of nodes become unavailable (e.g. because of a network partition). Consequently, Dynamo employs sloppy quorums instead of enforcing strict quorums: reads and writes still require R and W successful responses respectively, but those may include nodes that are not in the original designated N 'home' nodes [10].

When network interruption is over, any writes placed at temporary nodes by sloppy quorum are handed back to their designated home nodes by hinted handoff [10]. It is also possible for the hinted replicas to become unavailable before the hinted handoff. To prevent this possible unavailability of hinted replicas, and other such threats, from weakening the durability of the system, Dynamo uses an anti-entropy protocol to keep the replicas synchronized [10]. An anti-entropy protocol can be any background process that constantly looks for differences between replicas and copies any missing data from one node to another.

When compared with multi-leader replication, leaderless replication can offer better write availability. But leaderless replication is often ill-suited for services where conflicts are hard to reconcile, or for services where a stronger consistency model than causal consistency is required.

4.2 Handling Write Conflicts

Whenever there are two or more leaders acting concurrently on the same shard, there is the possibility of write conflict [19]. Consider Figure 1, where a meeting room for a particular date is being scheduled simultaneously by two users. Since the requests are simultaneous, the asynchronous replication can not reach either of the leaders before they apply their respective writes. Later, when the reservation is asynchronously replicated the conflict is finally noticed.

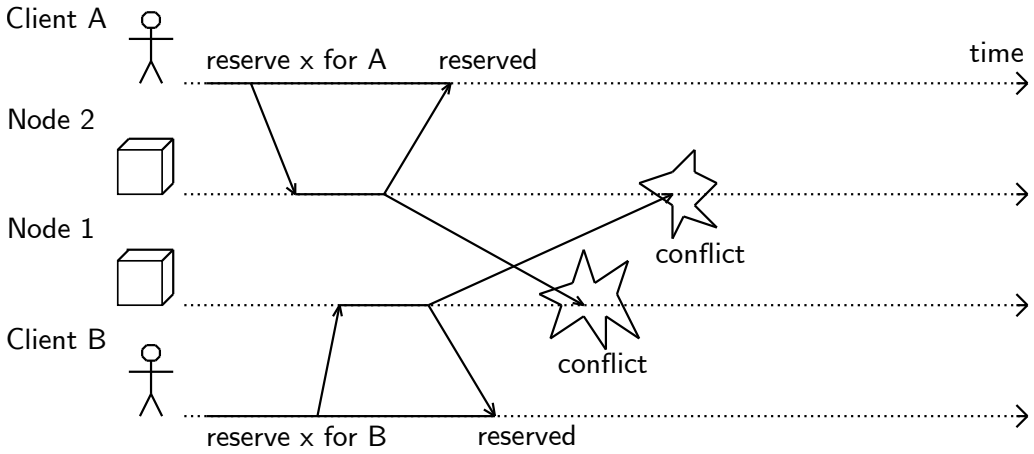

Figure 1: A write conflict caused by two concurrent writes.

Generally the best approach to write conflicts is to avoid them in the first place [19]. This can be as easy as choosing single-leader replication over the other replication strategies. But sometimes having multiple leaders to a shard is necessary. In such systems the possibility of conflict usually remains, but its likelihood can be reduced. For example, tightening the consistency requirements of the system can reduce the rate at which conflicts happen (see subsection 4.4). Another approach is to check if further sharding would better match the concurrency patterns of the system (see subsection 4.3). Furthermore, the allowed operations on the replicas can be designed in such a way that avoids conflicts altogether (see subsection 6.2). Even reverting back to synchronous replication, where conflicting writes are simply dropped should be considered [19].

Nevertheless, if there is even a slight chance of conflict, it will eventually happen [20]. Thus a robust conflict handling strategy is always needed to detect and handle write conflicts in a convergent way.

4.2.1 Detecting Concurrency

Not all concurrent writes are conflicts (e.g. the clients in Figure 1 might be reserving the same room for different dates), but conflicting writes most certainly happen during concurrent operation. It is thus essential to be able to detect concurrency in order to detect conflicts.

Strangely enough, detecting concurrency begins by trying to order events. For example, maintaining invariants usually requires some knowledge on the order in which operations occurred: a large withdrawal should only be possible after a sufficient deposit. The order in which we make deposits and withdrawals on our bank accounts should be clear to us (after all we are very synchronous), but usually it is not clear for the receiving system. In fact, the unpredictable delays between the components of a distributed system make it impossible to determine a natural total order for occurring events [19]. Therefore events are not ordered by natural total ordering, but by partial ordering in the form of **happens before** relations [21]. We say that two operations are concurrent when neither can be determined to be happened before the another [21].

Definition 3. Operation a_i happens before operation b_j when $i = j$ and a_i is submitted before b_j, or when $i \neq j$ and b_j is submitted after j has executed a_i, or when there exists an operation c_k, such that a_i happens before c_k and c_k happens before b_j,

where i and j are the sites that submitted operations a_i, b_j, and c_k respectively [21].

Definition 4. Operations a_i and b_j are concurrent if neither of them happened before the other [21].

Next we present some of the most important algorithms used to order operations in asynchronous environments. As a side note, when we speak about the 'size' requirement of an algorithm we mean its memory footprint, the amount of information required to be stored about the events to determine their order. In other words, the smaller the size requirement of an approach is, the better it performs. For example, less information attached to an operation means less network load and less ordering information to be inspected; moreover, smaller size requirement naturally means less wasted memory to store this information on each server.

Causal history. One way to detect 'happens before' relations is to attach to an operation all the identifiers of preceding operations [22]. This method is called

causal history. In causal history we determine if operation o_i happened before o_j by inspecting if o_i appears in the predecessors of o_j. Causal history has the advantage of being insensitive to the number of replicas, but its size grows with the amount of past operations [19]. In addition, for causal history to work properly, all operations have to be globally uniquely identifiable [23].

Version vectors. A somewhat newer, more popular way to detect 'happens before' relations is the use of version vectors. Version vectors are simply arrays of logical clocks. One clock for every source of concurrent events in the system [24]. Version vectors can be used to track causality in the system as follows [24]: First, each shard is accompanied with a version vector. Second, whenever a request is send to a shard, the sender must attach its related version vector to the request. Third, each time a node updates its local replica it increments its counter in the related version vector. Fourth, whenever an applicable request (a request is applicable if the version vector attached to the request is greater than or equal to the local vector in every element) to a replica is received, the receiver updates each element in the version vector of its local replica to be the maximum of the local and the received vector, and concludes by incrementing its counter by one, or, if the received request is not applicable the request is marked as concurrent and handled accordingly.

The downside of version vectors is that their size grows linearly with the number concurrently acting entities on the associated shard [23]. To illustrate, consider a multi-leader system, where clients can only modify their own data. In such a system the only source of concurrency are the leaders, which makes the size of version vectors same as the number of leaders. Unfortunately, it is far more usual for real world services to have concurrent clients, which increases the size of the version vectors to the number of clients.

Dotted version vectors. Further research on version vectors resulted in discovery of dotted version vectors [23]. This fairly new technique allows lossless representation of causality while only requiring an element per leader, even in the case of concurrent clients [23]. Although the improvement is potentially huge, the dotted version vectors differ only slightly from ordinary version vectors: whereas version vectors compress causal histories by representing only the last sequence number in a range of events, dotted version vectors are able to additionally represent individual events, that fall outside such ranges [23].

Lamport timestamps. Lamport timestamps are integers attached to replicas, used to indicate the number of operations applied to that replica [21]. Lamport timestamps are in fact predecessors to version vectors and are used very similarly, but with an important difference: instead of holding counters for every source of concurrency, Lamport timestamps hold only a single counter [21].

The algorithm to produce and to maintain Lamport timestamps works as follows[21]: First, every shard is timestamped and their replicas share that initial timestamp. Second, whenever a request to a replica is sent, the sender must attach its associated timestamp with the request. Third, whenever a node updates its local replica, it must increment the associated lamport timestamp by one. Finally, upon receiving an applicable request, the receiver must update the Lamport timestamps of the related replica to be the maximum of the original and the received timestamp. A request is applicable if the received timestamp is greater than or equal to the local timestamp.

The advantage with Lamport timestamps is their minimal size requirement, but this advantage comes with a cost: they can not capture causality. For example, even if we know that $t_a < t_c$ and $t_b < t_c$ we can not determine which of the operations a and b initiated the operation c by examining the timestamps t_a, t_b, and t_c alone.

Real-time clocks. Timestamps generated by real-time clock (RTC)s can also be used to order events in distributed systems and they do have one advantage over their logical counterparts [19]. Notably, RTCs can detect relations that happen via a "hidden channel" [21]. For example, an user submits an operation a with his laptop pc_a, walks over to an another computer pc_b to submit operation b, and continues on with his daily routines. From the user's point of view, it is obvious that a happened before b, but this might not be so for the underlying system receiving the operations. After all, computers pc_a and pc_b may not exchange any messages between the operations, rendering possible Lamport timestamps useless in ordering them. But if the operations were timestamped by RTCs (and the clocks sufficiently synchronised), it would be easy for the service to order the operations [21].

Unfortunately, small differences in their hardware causes RTCs to drift apart, making frequent synchronisations a necessity, and the synchronisation algorithms are not perfect either (see subsection 3.3). Thus we should take into consideration both the clock drift and the inaccuracy of the synchronisation algorithm, when using RTC timestamps. We denote the combined uncertainty of synchronisation and clock drift

by u. Because of this uncertainty u, it is wrong to timestamp an operation with an exact time t. Instead we should use uncertainty intervals $[t-u, t+u]$ as timestamps. Sadly most systems do not expose this interval to the use of the developers and instead only report the timestamp t [7, Chapter 8].

However, TrueTime in Google Spanner is an exception and explicitly reports the uncertainty interval of a local clock [25]. By reporting this uncertainty interval, spanner is able to correctly order events and detect concurrent ones. This follows from the observation, that if two events a and b have non-overlapping uncertainty intervals of $[a_{start}, a_{end}]$ and $[b_{start}, b_{end}]$ where $a_{end} < b_{start}$, then with certainty a happened before b. On the other hand, if these intervals had even a slight overlap, we are unsure which event happened first and thus conclude them to be concurrent. Spanner ensures causality of read-write transactions by waiting to the end of uncertainty interval before committing a transaction [25]. This wait ensures that any other transaction that may read the data is at sufficiently later time to avoid their uncertainty intervals from overlapping [25]. To mitigate this wait, Spanner needs to keep the uncertainty intervals as small as possible, and solves this by using multiple modern clock references (Global Positioning System (GPS) and atomic clocks) [25]. With these modern clock references Spanner has been reported to being able to synchronize clocks to within 7 ms in production [25].

4.2.2 Detecting Conflicts

Even equipped with the latest advancements in concurrency detection, we still need to weed out the conflicting operations from non-conflicting ones. After all the mere concurrency of two operations does not necessitate their conflict, but instead does indicate its possibility. Consider two concurrent requests incrementing a counter: if the requests contained the new state of the counter following the increment (same for both since they are concurrent), they would conflict, since the other increment would be lost; however, if the requests alternatively contained the change to the counter (e.g. +1) there would be no write conflict.

Semantic conflict detection. We say that an operation is in conflict when its precondition is unsatisfied, given the state of the replica after applying all operations preceding the conflicting operation [19]. For the system to verify the precondition of an operation, it needs knowledge on the semantics of the application the operation originated from. Such conflict detection is called semantic conflict detection.

Different applications have different notions on what it means to conflict; thus, in order for semantic conflict detection to work, it must provide the application layer the means to define what they consider as conflicting [26]. For example, Bayou, a weakly connected replicated storage system, uses dependency checks for automatic conflict detection [26]. These dependency checks are application specific routines attached to every write request submitted into the system. This enables application to indicate, for each write request, how Bayou should detect conflicts involving the write [26]. In more detail, a dependency check is an application supplied query to the state of the local replica accompanied with the expected result. If the check fails, the write is not performed and the related merge procedure is invoked.

Syntactic conflict detection. Conflict detection without semantic knowledge is called syntactic conflict detection [19]. In contrast to semantic conflict detection, all concurrent operations are considered as conflicting [19]. As an example of syntactic conflict detection, lets revisit the situation in Figure 1, where two users simultaneously schedule a particular meeting room, but this time for different dates. Lets assume the system uses version vectors to order operations. Thus, when the concurrency of the reservations is detected upon comparison of their version vectors, the system marks them (falsely) as conflicting and passes them over to conflict handling.

Because syntactic conflict detection lacks knowledge on the application semantics, it has the obvious downside of unnecessarily treating some actually non-conflicting operations as conflicts [19]. However, syntactic conflict detection might be desirable for applications with few concurrent operations, or if simple and generic solution is desired [19]. Moreover, syntactic conflict detection might sometimes be the only option because the needed semantic knowledge is simply unattainable (e.g. because of encryption) [27].

4.2.3 Resolving Conflicts

When conflicts occur, they need to be globally resolved, or the system risks becoming inconsistent. Conflicting writes always reach each other in different order at different nodes, and usually the 'later' write is the one interpreted as conflicting. This leads to a different write being interpreted as conflicting depending on the node. Thus, not only do we have to deal with the 'conflicting write', but with all writes in conflict to ensure one global order they are applied in. A carefully crafted conflict resolving strategy is thus needed in order for conflicting writes to eventually converge.

There are two approaches to conflict resolution: automatic and manual [19]. Automatic conflict resolution is usually done by an application specific routine which takes two (or more) versions of a replica and automatically merges them, creating a new version. In manual conflict resolution, the conflicting versions are presented to the user, who is then responsible to resolve the conflict and to resubmit the resolved replica back into the system. We are mainly interested in automatic conflict resolution, but as an example of an excellent use case for manual conflict resolution, we want to mention distributed version control systems, such as Git [28], where manual conflict resolution is used to solve merge conflicts.

Last write wins. Perhaps the easiest automatic conflict resolution strategy to implement is last-write-wins (LWW). In LWW potential conflicts are simply ignored by overwriting them with 'later' writes [19]. We wrote 'later', because after all we are talking about concurrent operations, meaning the system has actually no idea which of the writes is the 'later' one. However, we can arbitrarily order concurrent operations, by timestamping them at the receiving leaders, and by insisting that the one with the biggest timestamp is 'later' than the others. We want to stress, that these timestamps are only used to order operations deemed conflicting. The actual ordering of operations could be done e.g. by version vectors.

LWW is very simple to implement and adds very little overhead to the existing system [23]. Unlike many other automatic conflict resolution techniques, LWW does not require multiple versions of replicas to be stored, merge procedures, or writes to provide context. However, LWW has the obvious downside of being prone to lose data [7, Chapter 5]: whenever there are concurrent writes, only one of them will be applied, and others quietly discarded (even if they were reported as successes to their clients).

Lets revisit the meeting room booking example, where two clients are reserving the same meeting room for the same date, but this time with version vectors for conflict detection and LWW for conflict resolution. As was before, the clients simultaneously submit their reservations to different leaders but this time the leaders timestamp the writes with their local clocks. Later, when the version vectors of the writes are checked, the conflict is detected and resolved by keeping only the write with the most recent timestamp.

Conflict resolution in Bayou. Bayou expects applications to submit a merge procedure alongside each write for automatic conflict detection [26]. This mechanism permits applications to indicate for each write request, what steps should be taken to resolve any conflicts found during the related dependency check [26]. In more detail, if the accompanying dependency check fails, the write is not performed and the related merge procedure is invoked. This merge procedure is a general program, written in a high-level, interpreted language [26]. The responsibility of this procedure is to resolve any conflicts found during the dependency check and to produce a revised update to apply [26]. Finally, if the merge procedure itself fails, the error is logged and handed over to manual conflict resolution [26].

Lets revisit the meeting room example again, but this time with conflict detection and resolution strategies similar to the ones used in Bayou. The reservations are still concurrent and conflicting, but this time the clients pass a dependency check and a merge procedure alongside their requests. Considering the nature of the request, the dependency check should be a query that checks if the meeting room is available for the date the client is trying to reserve. Similarly, the merge procedure could be a collection of reservations for other dates and other meeting rooms, selected by the client as secondary choices in the case of a conflicting reservation. Now with each request being accompanied with these checks and procedures, the leaders can just apply the requests and later when the conflict is detected apply the merge procedure of the conflicting write. But it is important to notice that the conflicting reservation is different at different leaders depending on the order in which these reservations are applied. This leads to inconsistencies since each leader applies the merge procedure of a different reservation. Thus, these writes are treated as tentative until they have been numbered by a central authority issuing globally monotonically increasing identifier (ID)s. When a leader learns the ID of a reservation it has applied it reapplies all the tentative reservations it is holding according to their IDs. A leader commits a tentative write only after it has received and applied all the writes preceding the tentative write. This way all leaders are guaranteed to eventually apply all reservations in the same order leading to a consistent system.

Conflict resolution in Bayou is very powerful but it does incur some overhead. Due to the merge procedures, Bayou can use application specific knowledge when resolving conflicts and is thus able to solve conflicts that occur either far away from the user or long after the user has gone offline [26]. For example, when requesting a meeting room for a particular date, the application could ask the user if some another date would also suffice in the event of a double booking. Because of these secondary dates,

the user does not have to wait for the booking to fail or succeed: whenever a conflict occurs, Bayou simply applies the merge procedure on the conflicting replica. Being this flexible adds some overhead: the combined length of the dependency check and merge procedure can be many times the length of the actual write itself [26]! In addition, these procedures have to be invoked every time a write is received or replicated, adding some computational overhead.

Read repair in Dynamo. Amazon's Dynamo uses a technique called read repair to detect and to resolve conflicts [10]. Like was already discussed, Dynamo uses quorum reads and writes to ensure consistency across its replicas. Depending on how these quorums are configured, a write might not reach all of the replicas. Thus, at any given moment there might be multiple versions of a replica inside a Dynamo instance [10]. It is entirely possible for these differing versions to further diverge if the quorums are configured to low enough values. Dynamo solves this issue by a technique called read repair. Whenever multiple versions of a replica arise during a read, dynamo tries to syntactically reconcile the versions if they are causally related. If the versions are no longer causally related, Dynamo passes them with the corresponding version context to the client application for reconciliation [10]. Finally, when the client has reconciled the conflicting versions, its next write will cause the divergent branches to collapse into a single one [10].

Let us revisit the meeting room example once more, but now with leaderless replication. Let us assume there are N replicas, one of which is the coordinator, and R and W quorum requirements for reads and writes respectively. To indicate the state of the meeting room booking system the client is aware of, they pass alongside each reservation the version vector they received with the last read. The coordinator replica handles all requests: it updates the version vector of received reservation, writes the reservation locally, and finally passes the reservation (along with its updated version vector) to the rest of the replicas. As soon as $W - 1$ of the replicas respond the write is considered successful. If a node receives a reservation that can be deemed causally related to its local state, it applies the reservation and discards the old version, and if not, it stores both versions. These conflicts are then reconciled by read repair: either syntactically at the coordinator node (if the different versions prove to be causally related) or semantically at the client. This means that a client wanting to be sure of their reservation, would have to check if their reservation was successful, and if not, solve the conflict (e.g. by picking another date or room) and write the reconciled state back to the system.

4.3 Sharding

Sharding is the act of breaking down a dataset into more manageable, non-overlapping pieces called shards. To understand the role of sharding in distributed systems let us consider how it complements replication. First off, sharding designates the unit of replication: when deciding on how many shards a dataset should be divided into, we actually decide on the replica sizes. Having too few shards causes replicas to be large, which can translate into the nodes reaching their compute or storage capacity. Sharding thus complements replication by making existing replicas more manageable [6]. Of course reducing existing shard sizes increases their number, requiring (enabling) new nodes to be added into the system as replica groups to handle the newly introduced shards.

Sharding improves scalability. Since shards do not overlap, a request to a shard can be handled in isolation from others. This isolation directly translates into improved scalability. For example, consider a shard that has only one leader and multiple followers. Scaling against the read throughput of such a shard is easy: all that is required is to add more followers to share the load. But adding more followers does not help in scaling against the write throughput. More leaders are needed instead. But the application at hand might be such where concurrent writes to a shard are not easily handled. This is where the isolation of shards comes in to play: by further breaking down the original shard, these new shards can be governed by new leaders in isolation of each other. Indeed, further sharding reduces traffic on any given shard by a factor of N, where N is the number of new shards [29].

Sharding also improves availability. Since shards are isolated, the unavailability of one does not affect the availability of others [30]. For example, if a network partition caused every replica of a particular shard to become unavailable, only that part of the dataset would in turn become unavailable.

Sharding can be used to improve performance by reducing write conflicts. A good sharding strategy not only looks into the shard sizes, but into the applications access patterns: by isolating concurrent operations to different shards, sharding can reduce the rate at which write conflicts occur [19]. In fact, sharding can be seen as a specific case of a more general pattern: coordination avoiding (see section 6). For example, consider a multi-leader system, which suffers from poor performance because of high-conflict rate caused by too many concurrent users per shard. One possible solution could be to reduce the amount of users acting concurrently on any one shard by further sharding. If possible, we could take this approach to the extreme,

and assign every user to their own shard (effectively turning the multi-leader system into a single-leader one) cancelling out any possibility of a write conflict.

But sharding also has its downsides. Transactions across several shards might be hard to reason about: if a transaction succeeds in one shard but fails in another, what will follow? Consequently, some popular systems forbid cross-shard transactions entirely [10, 31]. Another downside of sharding is the additional need for request routing [7, Chapter 6]: when every node is no longer a copy of the others, we have to have some system in place to route requests to their respective shards. Lastly, sharding requires occasional rebalancing. After all, as time passes things change: request throughput fluctuates, some shards grow, some may diminish, and some replicas become unavailable. Rebalancing might require operator intervention (manual rebalancing) or additional mechanisms to be implemented for automatic rebalancing.

4.3.1 Effective Sharding

Choosing the right sharding strategy starts by defining what constitutes a shard. For instance, Google File System (GFS) [11] uses file chunks as shards (chunks are equivalent to blocks used in ordinary filesystems). Other services, such as Dynamo [10], BigTable [31], and COPS [32] use ranges of key-value pairs as shards. In contrast, PNUTS [33] uses simplified relational database tablets as shards.

Shards should be large enough to avoid the overhead of having to manage many small shards. For example, GFS uses 64 MB chunk size, which is much larger than the default block size used in ordinary file systems (often 4 KB), to reduce overhead at the master node [11]. Because of this large chunk size clients need to interact less with the master node. In GFS, to operate on a chunk, client only needs to know the location of the chunk. Only if the location is unknown, does the client ask for it from the master. Thus all subsequent operations on a given chunk require only one initial request to the master [11]. This is where the large chunk size comes to play: since chunks are large, it is more likely that clients need to interact less with different chunks, and thus, less with the master [11]. Furthermore, this large chunk size enables clients to cache all the required chunk location information for a multi-TB working set [11]. Therefore, network bandwidth is spared for actual read/write traffic and the master can concentrate on its duties. Lastly, the large chunk size means less chunks, which translates into less metadata, enabling master to keep it all in memory [11].

While shards should be large to avoid overhead, they should simultaneously be small enough to avoid unnecessary hot spots and write conflicts. For instance, the large chunk size used in GFS can lead to hot spots when dealing with many small files. Because of the large chunk size, small files consist of only a few chunks, usually just one. Consequently, if many clients become interested in some small file, it is highly likely that their requests concentrate on only a few nodes. Depending on the intensity of this traffic, these nodes might become hot spots [11]. Moreover, an unnecessarily large chunk size can lead to increased conflict rate [19]. This happens because number of requests to a shard is roughly inversely proportional to the number of shards in the system; thus a few large shards tend to receive more concurrent requests than many small ones [19, 29].

4.3.2 Sharding Key-Value Data

So we now understand the benefits of sharding and have an idea on how shard size affects the system, but we are still missing a key piece of information: how to decide which data to store on which nodes? The goal is to spread the data and the related traffic evenly across the system [7, Chapter 6]. A defective (or otherwise ill-fitted) sharding strategy can lead to skewed data distribution, where some shards receive more data or query traffic than others. In the worst case scenario, all of the load could end up in just one shard, causing rest of the nodes to idle. Shards with disproportionate load are called hot spots [7, Chapter 6]. To avoid these hot spots, we have to be aware of how data is accessed and pick the sharding strategy accordingly. If the data is accessed in ranges, key range sharding should be considered, and if not, hash sharding should be chosen.

Key range sharding. In key range sharding, a shard is assigned all keys from one minimum value up to some maximum; the next shard continues from there [7, Chapter 6]. Each shard is thus a sorted range of keys, which has the advantage of making range scans efficient and easy to implement [7, Chapter 6]. This approach, however, may lead to hot spots if the application is only interested in certain ranges [7, Chapter 6]. For example, suppose an application is only interested in the latest data, and the data is key range sharded by timestamp. Since the application is only interested in the latest data (which may reside in a single shard), we risk nodes holding the replicas of this shard becoming hot spots.

Google's BigTable is a good example of a system that employs key range sharding

for great results. BigTable is a sparse, distributed, persistent multidimensional lexicographically sorted map [31]. A BigTable cluster holds a number of tables, which consist of tablets. Each tablet contains all data associated with a row range. Initially each table consist of just one tablet, but as a table grows, it is automatically split into multiple tablets, which are then distributed among tablet servers [31]. An interesting feature of BigTable is that row keys can be arbitrary strings (up to 64 KB in size), which paired with the lexicographical ordering gives an application freedom to have their data stored in a way that matches its access patterns [31]. For example, Google Earth uses BigTable to serve high-resolution satellite imagery of the surface of the world. Google Earth utilises the lexicographical row order of the BigTable by using row keys that put adjacent geographic segments near each other [31].

Hash sharding. Like in key range sharding, shards in hash sharding constitute of rows of keys, but with an important difference: each key is hashed before assignment [7, Chapter 6]. This seemingly tiny difference makes hash sharding effectively the exact opposite of key range sharding: the hashing ensures that similar keys are no longer stored in near vicinity of each other, defending against hot spots [7, Chapter 6]. As a consequence, all possible order the original keys may have had is destroyed (a good hash function turns adjacent keys into completely different ones!), making hash sharding a poor choice for applications interested in ranges of keys.

Amazon's Dynamo, a highly scalable, always writeable, key-value store, is a perfect example of successful application of hash sharding for great results [10]. Dynamo was designed as a primary-key access store, because that was the query model many of Amazon's core services could work with [10]. Because even the slightest outage has significant financial consequences and impacts customer trust, Dynamo was designed to satisfy very stringent service-level agreement (SLA), requiring it to stay virtually always available [10]. To satisfy such a SLA, an even distribution of keys was absolutely necessary and thus hash sharding was chosen as the sharding strategy for Dynamo.

To be more specific, Dynamo employs a hash sharding technique called consistent hashing to minimise the impact of a departing node. At Amazon's scale, some part of their network and server components is always failing at any given moment [10]. This means frequent reassignment of keys from failing nodes to available ones. To minimise the performance impact of node departures, consistent hashing [34] was

chosen as the basis for the sharding strategy used in Dynamo, since it restricts the impact of node departures to their immediate neighbours [10]. This sharding strategy works as follows: the hash function maps each key to a point on an edge of a ring, each node is similarly mapped to the same ring, and finally each of the mapped nodes is assigned with the keys between it and its preceding neighbour [34]. Thus every node is responsible for an arc on the ring. Dynamo extends consistent hashing by using virtual nodes instead of physical ones to better suit its heterogeneous network of nodes [10]. Each physical node is assigned virtual nodes according to its capacity, which are then assigned to random positions on the ring.

4.4 Consistency

Every time a leader accepts a write, the introduced changes become available at the followers only after successful replication has taken place. Depending on the implementation this could be almost instantaneous, or take an indefinite amount of time [26]. Therefore, if we were to stop a distributed system, we would be very likely to see inconsistencies among its replicas. These inconsistencies are always present regardless of the replication strategy used [7, Chapter 9]. For such a system to appear consistent, some kind of an abstraction is required between replicas and request handling. We call this abstraction the consistency model. There are many different consistency models offering different consistency levels, meaning how consistent the system appears to be. In the rest of this subsection, we present a handful of consistency models, picked solely to support later discussion on distributed system trade-offs in section 5. The consistency models presented are in rough order from strongest to weakest, but slight deviations from that order were made, because some of the models are only sensible to discuss after others. The actual order is: linearisability, sequential consistency, causal consistency, session based consistency, and finally eventual consistency.

4.4.1 Linearisability

We say that a set of concurrent operations is linearisable if its result is equivalent to a legal sequential computation of that same set [35]. In other words, all operations should appear to take effect instantaneously and the order of non-concurrent operations should be preserved [35]. Consequently, the outcomes of concurrent operations are restricted: if all operations should appear to take effect instantaneously,

concurrent operations have to be arbitrarily ordered.

To better understand how linearisability restricts outcomes of concurrent operations, consider the request diagram depicted in Figure 2. In the figure, each bar is a request made by a client, where the start of a bar indicates the sending, and the end the receiving of the corresponding response. Notice how in this request diagram, the reads concurrent with the write may return either the new or the old value. Thus the operations do not appear to take effect instantaneously, and we can conclude this system to not be linearisable.

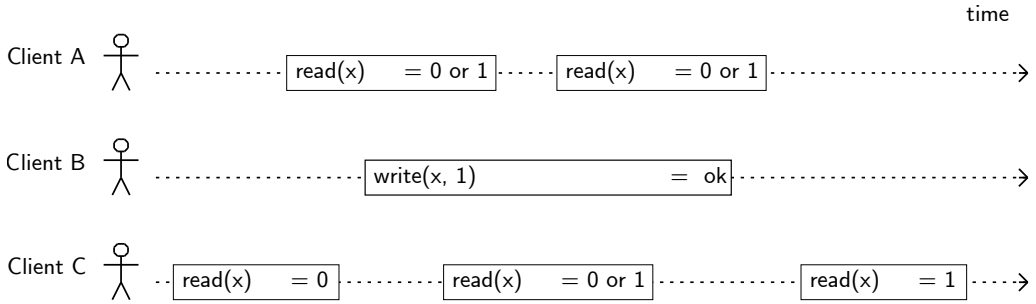

Figure 2: A read request concurrent with a write may return either the new or the old value.

In contrast, consider the request diagram of Figure 3, produced by an another service. In this figure, as soon as the first read returns the new value all the subsequent reads must do the same, until another write takes place. Thus the operations appear to take place instantaneously and we can determine this service to be linearisable.

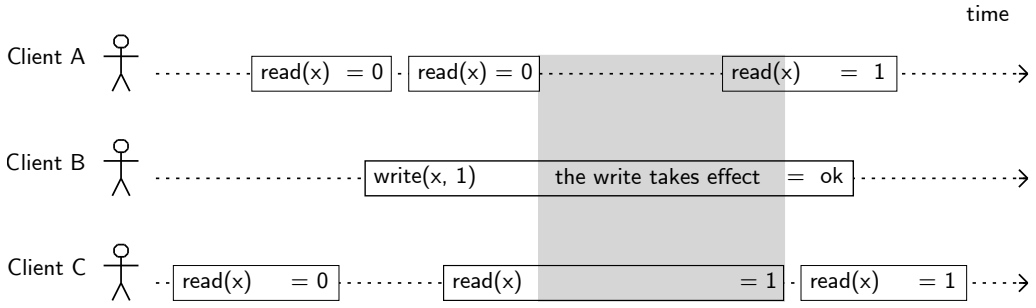

Figure 3: After a read returns the new value, all subsequent reads must do the same.

Implementing linearisability from scratch is difficult work: we would be first required to come up with a fault-tolerant consensus protocol (see subsection 4.5), which is a feat in itself. However, several such protocols already exist and it would be best to

leverage one of them. As soon as the consensus protocol is in place, linearisability can be implemented on top of it by voting for the global order in which each arriving request is to be served. One way to achieve this global service order is by proposing each received request until they have been accepted with consensus.

Linearisability is implemented through rounds of fault-tolerant consensus (see subsection 4.5). Each request is proposed to be served, but is only applied after its place at global order of operations is decided upon. Sounds simple, but there is actually a lot going under the hood: before a request can be applied, it first needs to reach at least the majority of the replicas, which then have to reach a decision together. Hence, a lot of communication is required to apply even a single request, which brings us to the downsides linearisability.

Linearisability suffers of poor throughput, response times, and availability. Every one of these downsides is a direct consequence of the high coordination requirements of the underlying consensus protocol. First, since a request can be served only after the majority has decided so, the availability of each node depends on it being connected to at least the remaining majority (see subsection 5.2). Second, because requests are effectively serialised (their order is globally decided upon), throughput is limited to serialised execution. Third, since serving each request requires communication between at least the majority of replicas, response times are at best proportional to the round-trip time (RTT) to the slowest responding replica.

4.4.2 Serialisability

Concurrent execution of a set of transactions is serialisable if there exists an equivalent serial execution of the said set [36]. This is important because serial execution is always conflict-free. In other words, serialisability is a guarantee against conflicts: whenever we can find a serialisable way to concurrently execute some transactions, we are additionally guarded against conflicts. Serialisability, however, does not impose real-time ordering constraints [36]. Nor does it imply any deterministic order [36].

Simplest way to implement serialisability is through actual serial execution [7, Chapter 7]. In serial execution, each shard is assigned a dedicated CPU core, through which all of its transactions must be executed [7, Chapter 7]. Interestingly, serial execution has only recently become a viable way of providing serialisability [37]. But to keep throughput acceptable, all transactions in these systems have to be kept

small and fast to avoid occasional slow ones from stalling entire shards [7, Chapter 7]. Similarly, cross-shard transactions should also be avoided to prevent slow cross-shard transactions from causing stalls across multiple shards [7, Chapter 7]. In conclusion, serial execution has become a viable way to provide serialisability, but these implementations rely on the transactions being small and fast in order to provide good throughput.

For many decades there was only one widely used method to implement serializability: two-phase locking [7, Chapter 7]. In two-phase locking, every object is assigned a lock, which has to be acquired before access to the object is granted. Concurrent reads are made possible by allowing read transactions to acquire locks in shared mode, but write transactions are required to acquire locks in exclusive mode [7, Chapter 7]. These exclusive mode acquires work as follows: if the lock is already acquired in shared mode, the transaction holding the lock in exclusive mode has to wait for the reading transactions to release the lock; additionally, if another transaction tries to acquire the lock, while it is already held in exclusive mode, it has to wait for the lock to be released. Serialisability implemented through two-phase locking has the advantage of concurrent reads, but its throughput is hindered by the lock acquiring and releasing overhead. Another great disadvantage of two-phase locking is the overall reduced concurrency: whenever two transactions try to do something that might end up in a race condition, one of the transactions is forced to wait for the other one to complete [7, Chapter 7].

Lastly, serialisability can be implemented using Serialisable Snapshot Isolation (SSI), a fairly new extension over Snapshot Isolation (SI) [38]. To understand SSI, its best to start by understanding SI. The core idea behind SI is the snapshot: whenever a write commits it creates a snapshot, a timestamped version of the modified data. These snapshots are then used to isolate reads from writes. Each received transaction is assigned a timestamp, that is used to determine which snapshot the transaction should see. Consequently, a transaction may not see the most recent version of data, but instead it sees a version that was written by the last transaction to commit before its timestamp [38]. In addition, SI enforces a restriction called “First-Committer-Wins” which prevents transactions from modifying data if another concurrent transaction has already modified it and commited. But SI by itself can not be used to implement serialisability, because it can not guarantee all executions to be serialisable [39].

Now that we understand SI, SSI is fairly simple to explain. SSI is like SI but with

added book-keeping that enables it to dynamically detect and abort transactions where a non-serialisable execution could occur [38]. This is achieved by maintaining a serialisation graph and by looking for certain anti-dependency [40] patterns between on-going transactions [38].

When compared with two-phase locking and serial execution, SSI seems to have the least drawbacks and would thus be our choice to implement serialisability. The advantage of SSI over two-phase locking is that in SSI one transaction does not have to wait for locks held by another transaction. Compared with serial execution, the advantage of SSI is that its throughput per shard is not limited to a single CPU core. SSI does, however, have its own problems. The performance of SSI is significantly affected by the number of transaction aborts and by the overhead resulting from having to maintain the dependency graph [7, Chapter 7]. SSI also requires read-write transactions to be fairly small and fast, since long running read-write transactions are likely to run into conflict and abort [7, Chapter 7]. Like two-phase locking, SSI suffers from poor availability: to apply a transaction, a node has to be connected to majority of the replicas to be aware of other transactions in the system. On the other hand, in serial execution a node only needs to stay connected to one node (the one with the dedicated CPU core of the shard) to continue to accept writes, but this approach introduces a single point of failure (what if the node holding the dedicated core becomes unavailable?).

4.4.3 Sequential Consistency

Sequential consistency requires execution of concurrent operations to be equivalent to some serial execution, which order is consistent with the order seen at individual processes [41].

To understand sequential consistency it is best to compare it with linearisability. Firstly, linearisability is more convenient to use because it preserves real-time ordering of operations, and hence it corresponds more naturally to the notion of atomic execution of operations [35]. In contrast, sequential consistency only requires the order to be be consistent with the views of individual processes, which may differ. Secondly, linearisability is composable, whereas sequential consistency is not: if operations on each object are linearisable, then all operations in the system are linearisable [35]. Thus we can conclude sequential consistency to be considerably weaker than linearisability. In fact, it is just weak enough to be implemented with reduced coordination either on writes or on reads (see section 5).

Sequential consistency can be implemented with reduced coordination on either writes or reads to such an extent that the chosen class of transactions can return immediately [42]. For example, fast reads can be achieved by instructing each node to hold a local copy of every replica against which the reads are to be performed. This enables reads to return immediately, but to ensure sequential consistency writes must be implemented to be linearisable. The usual approach is to use atomic broadcast [43] for writes. Atomic broadcast works as follows: each broadcasted write is marked as pending, replicas apply every write they receive by atomic broadcast, and finally, pending writes are acknowledged to their clients only after the node receives its copy from the atomic broadcast service. Implementing sequential consistency with fast writes is very similar, but with the appropriate changes [42].

Albeit being weaker than linearisability, sequential consistency is still a relatively strong consistency guarantee. Depending on the implementation it can provide high availability and throughput for either writes or reads, but not for both (see subsection 5.5). It is a solid choice for services with a disproportionate read/write ratio since the overall response times can be lowered by choosing an implementation that correctly reflects this ratio. For example, let us assume a system, which receives read-heavy query traffic, e.g. roughly 10% of requests to this system are writes. Let us further assume that in this system a linearisable operation takes 2 seconds to complete on average. Now if the system was completely linearisable, the average operation completion time would be 2 seconds. On the other hand, if the system was sequentially consistent we could choose between linearisable reads and writes. Choosing linearisable reads would make the average operation completion time to be 1.8 seconds $((90 * 2s + 10 * 0s)/100 = 1.8s)$. Similarly, choosing linearisable writes would make the average operation completion time to be 0.2 seconds $((10 * 2s + 90 * 0s)/100 = 0.2s)$. Thus, for this particular example system, the average operation completion time can be lowered from 1.8 seconds to 0.2 seconds just by selecting the sequential consistency implementation according to the read/write ratio of the query traffic.

4.4.4 Causal Consistency

In causal consistency, all potentially causally related operations are guaranteed to be seen by every process in the same order [44]. But there is no such guarantee for concurrent operations, which may be seen to take place in different order between different processes [44]. Interestingly, causal consistency requires so little

coordination, that it can return instantly for both reads and writes. It is in fact the strongest consistency model that can stay available for reads and writes during network partitions (see subsection 5.5).

In causal consistency implementations each node in the system maintains a set of writes, which can be safely locally modified and read from [45]. Whenever a new remote write arrives from another node in the system, its metadata is checked to ensure that its causal dependencies are satisfied by the local set of writes [45]. If the dependencies are satisfied, the agent applies the write, and if not, the node waits until the missing dependencies have been applied [45]. A write becomes visible only after it has been applied to the local set of writes, against which reads are performed. To ensure that clients can always view their latest writes, despite the replication lag, clients should be assigned with a preferred node, through which their requests are always spread into the system. This way there is always an 'up-to-date' node in the system from each client's perspective.

Although causal consistency can offer immediate return for both writes and reads, it still has some downsides, like the trade-off between write throughput and the time it takes for these writes to become visible [45]. This trade-off exists because causally consistent replicas have to defer from applying writes until their dependencies have been applied, but the replicas can only apply these missing dependencies according to their individual throughputs [45]. Hence the trade-off: an increase in write traffic causes an increase in the time it takes for these writes to become visible.

As an another downside, causal consistency does not scale well against throughput [45]. For example, consider a causally consistent system of two replicas with equal apply-capacities of A. Under normal operation the traffic at each replica should be a mixture of new and replicated writes, with the combined throughput staying below A to avoid operations from queueing up. In other words, to avoid operations from queueing up, the aggregate write throughput of the system should be limited to the apply-capacity of its poorest replica (A in this case)! Thus, to scale out a causally consistent system, each replica should be scaled up accordingly [45]. More precisely, scaling the number of replicas from N to M requires upwards scaling of replicas by $\mathcal{O}(M^2/N^2)$ in apply-capacity [45].

4.4.5 Eventual Consistency

The only guarantee of eventual consistency is that if new writes stop arriving each replica will eventually receive them all [46]. Eventual consistency does not make any ordering guarantees. Instead, an eventually consistent system can show states produced by any possible subset of submitted operations [46].

To understand how serious this complete lack of ordering guarantees is, consider an eventually consistent system with just a single client making a single request. Now, on subsequent reads the client may sometimes see the effects of the write and sometimes not. The result depends on if the read reaches an up-to-date node or not.

Despite its weaknesses, eventual consistency can sometimes be desirable because of its high throughput. Moreover, an eventually consistent system can stay available as long as a single node in the system stays available. Even its lack of ordering guarantees can be alleviated by careful design (see subsection 6.2), but this usually limits the type of operations the service can accept.

4.4.6 Session Based Consistency

Session based consistency was designed to solve the lack of ordering guarantees of eventual consistency while still mimicking its great performance and availability properties [47]. The core idea behind session based consistency is to provide each client a view of an inconsistent system that is still consistent with the client's actions (session) [47].

Session based consistency comes in four different guarantees: Read your writes (RYW), Monotonic reads (MR), Writes follow reads (WFR), and Monotonic writes (MW). RYW guarantees that effects of any writes are visible to later reads within a session [47]. MR guarantees that consecutive reads show the state of the system to be increasingly up-to-date; a read may never return a staler result than some previous read [47]. WFR guarantees that new writes are ordered after any writes whose effects were seen by previous reads within the session [47]. This guarantee differs from RYW and MR in that it spans outside a single session: the order of writes within some session is guaranteed to be same for other sessions as well [47]. Finally, MW simply guarantees that within a session new writes are applied after earlier ones [47].

Session based consistency guarantees can be implemented with very little added

coordination [47]. In session based consistency, each write is assigned an identifier, and every request is required to include identifiers of relevant writes within the session [47]. The actual session guarantees are then provided by a session manager, which basically chooses what replicas to pass client's requests to [47]. One possible location for the session manager is the client stub that the client uses to connect to available servers. This way the session manager can provide the RYW and MR guarantees by connecting the client only to replicas that advertise having seen the writes with the required identifiers [47]. The WFR and MW guarantees can be implemented by further requiring the replicas to order new writes after old ones and to preserve the write order during anti-entropy (anti-entropy is the act of seeking and updating stale replicas; usually done in the background by a dedicated process) [47].

Session based consistency offers high-availability, scalability, and disconnected operation, while being relatively simple to implement [47]. Session based consistency is also very flexible because of its four different consistency guarantees that can be balanced accordingly to the characteristics of each operation. Consequently, an application build on top of session based consistency can select different consistency requirements for each type of operation [47]. It is even possible to combine several of these guarantees if needed [47].

4.5 Fault-Tolerant Consensus

Reaching agreement among remote processes is one of the most fundamental problems in distributed computing [48]. This is called the consensus problem and is normally formalised as follows: although each node is free to propose any action (e.g. an operation it wants to apply), they still must decide on one of the proposed by consensus [7, Chapter 9].

A fault-tolerant consensus algorithm must satisfy the termination, validity, integrity, and agreement properties [49, Chapter 5]. By termination we require all correct processes to eventually decide. The validity property is necessary to enforce non-triviality: processes may only decide on proposed values. By integrity we mean that all processes must decide on at most one value. Lastly, by agreement we require all processes to decide uniformly: no two processes decide differently.

Agreement and integrity properties together form the heart of the consensus algorithm, while termination and validity ensure its usefulness. If we only required agreement and integrity from a consensus algorithm, the result would be an algo-

rithm where all processes would decide uniformly on some single value. But by requiring only agreement and integrity, we can not guarantee this algorithm to ever reach a decision. Some of the processes may stall indefinitely. Hence, by requiring termination we ensure that decision is eventually reached. But this decision might be anything. For example, we could trivially satisfy agreement, integrity, and termination requirements by an algorithm devised to always decide on some fixed value. So by further requiring validity we rule out such trivial solutions.

4.5.1 Applications

Some consensus algorithms, such as Raft [50] and ZooKeeper Atomic Broadcast (ZAB) [51], do not directly reach consensus by deciding on individual proposals, but instead decide on the order in which the proposed operations are applied [7, Chapter 9]. In other words, they solve total order broadcast, in which messages are delivered exactly once, in the same order, and to all nodes [52]. Total order broadcast is equivalent to fault-tolerant consensus, since either can be implemented on top of the another [53]. For example, total order broadcast can be implemented on top of rounds of fault-tolerant consensus, where in each round nodes may propose messages to be send and by consensus decide on the order they are received [53].

In subsubsection 4.4.1 we briefly remarked on how linearisability requires fault-tolerant consensus. To be more exact, linearisability implementations often rely on total order broadcast. Conceptually it is rather simple: every received request is first sent by total order broadcast to all other replicas and is only applied after the node receives it back by total order broadcast. This way all nodes have seen and accepted the request and its position in the global order of operations.

In order to avoid the so called split-brain problem, single-leader replication implementations have to ensure that at most one of the replicas can be leader at a time [7, Chapter 5]. In other words, the replicas have to elect the leader by consensus: whenever the connection to leader is lost, the followers start to propose leader candidates and by consensus decide on one.

4.5.2 Implementations

In subsection 4.1 we discussed how single-leader replication limits accepting of writes to a single node, the leader. Nevertheless, this also means that the leader decides on the order in which writes are applied and replicated, a situation effectively equivalent

to total order broadcast.

But if single-leader replication is essentially total order broadcast, which is itself equivalent to fault-tolerant consensus, does this mean that fault-tolerant consensus can be simply implemented on top of single-leader replication? Unfortunately, the answer is no. Leader election would still have to be done through fault-tolerant consensus [7, Chapter 9]. So in order to elect a leader, we would need a leader to begin with.

Many fault-tolerant consensus implementations use internally a leader in some form or another, but they make no guarantees of the uniqueness of the leader [54, 50, 51]. Rather, they are able to make a weaker promise. The operation of each of these algorithms is divided into epochs, and inside each epoch the leader is guaranteed to be unique [7, Chapter 9].

Each epoch is assigned an epoch number which are monotonically increased by each internal leader election. A leader election is held whenever the leader is thought to be dead. Just like in single-leader replication, the successor is selected by vote, but is additionally assigned with the epoch number of the election. To guarantee the monotonic increase of epoch numbers, the new epoch number is always chosen so that is larger than any other known epoch number. This works since in order for an election to succeed the majority has to be present, and therefore at least one of the nodes must have participated in the latest successful election and thus know the previous largest epoch number. It is important that the epoch numbers are monotonically increasing, since they are used to solve conflicts between leaders of different epochs [7, Chapter 9].

To prevent leaders of old epochs from introducing conflicts, they are required to confirm their leadership by vote before each decision [7, Chapter 9]. This works by the leader proposing its epoch number before each decision [7, Chapter 9]. The leadership is then confirmed if the leader does not learn of a higher epoch number from the majority. The followers only accept proposals if they do not know about a leader with a higher epoch number.

The uniqueness of leader between epochs is hence guaranteed by two kinds of voting: leader elections and proposals [7, Chapter 9]. The key insight is that at least one node in each successful vote is guaranteed to have participated in the previous successful vote since all successful votes require majority. Hence, in every vote there either is at least one node which knows the previous epoch number, or the vote fails because it did not reach the majority. Thus, even though the leader might

sometimes change, it is guaranteed to be unique between epochs.

4.5.3 Summary of Fault-Tolerant Consensus

Fault-tolerant consensus is an important breakthrough for distributed systems, since it can guarantee termination (if at least majority stays available) and concrete safety properties (agreement, integrity, and validity) despite of the unreliability of the underlying system [7, Chapter 9]. Without fault-tolerant consensus there would not be total order broadcast, nor linearisable systems. Even guaranteeing the uniqueness of the elected leader, a fundamental need in single-leader replication, would be impossible.

Fault-tolerant consensus, however, comes at a cost. First, during a network partition, only the majority of the nodes can remain available for users. Second, in environments with highly variable network delays, it often happens that nodes start unnecessary elections, when they (falsely) believe the leader to have failed [7, Chapter 9]. This can lead to performance problems since the system can end up spending more time electing a new leader, than doing actual work. Third, voting for every decision requires a lot of coordination, which can be too costly for systems that aim to offer high-availability or quick response times.

5 Trade-offs and Impossibility Results

Designing a distributed system includes surprisingly many trade-offs one has to be aware of. In a world full of failures, it is simply impossible for a distributed system to be highly available, strongly consistent, and partition tolerant at the same time.

We start by examining one of the most important distributed systems impossibility results, Fischer's, Lynch's, and Paterson's impossibility proof (FLP), in subsection 5.1. Next, in subsection 5.2 we move onto the Brewer's theorem (CAP) theorem and explain why it is impossible for a distributed system to be simultaneously consistent, available, and partition tolerant. Then, in subsection 5.3 we discuss Abadi's extension to the CAP theorem (PACELC) an extension to CAP theorem. Next, in subsection 5.4 we examine how designing for graceful degradation enables us to build systems that do not have to choose either consistency or availability over the another, but to make this choice on a more continuous axis, balancing the harvest and yield of the system. Lastly, in subsection 5.5 we finish by looking into delay-sensitivity framework, which provides a tool to examine how tightly systems are coupled to changes in network delay.

5.1 FLP

How many process failures can a completely asynchronous consensus protocol tolerate? None, states the FLP impossibility result, one of the most important results in distributed systems theory [48].

The proof of the FLP is built upon a set of non-assumptions on the system model. The system is assumed to be completely asynchronous: there are no assumptions on relative speeds of processes, on synchronised clocks, on how long messages can be delayed, nor on the order in which they might arrive [48]. Consequently, it is impossible for a process to tell if another is dead, or advancing really slowly [48]. They continue to give this impossibility result even more weight, by assuming messages to be delivered correctly and exactly once [48].

The proof for the FLP result is rather complicated and out of this thesis' scope, but we still wish to present its outline. As the system is assumed to be completely asynchronous, messages may get arbitrarily reordered, and therefore there exists many different possible runs for a single initial configuration of messages [48].

Additionally, for every consensus protocol (that can supposedly tolerate a single

fault) there exists some initial configuration of messages, where the outcome of the system is not determined by the configuration but by the order in which the messages are received (first lemma: there exists a bivalent initial configuration) [48].

Furthermore, if a consensus protocol starts from a bivalent configuration and we delay a message applicable to that configuration, then the set of configurations reachable through any sequence of messages where the delayed message is applied last contains a bivalent configuration (second lemma) [48].

A consensus protocol is partially correct if each reachable configuration has at most one decision value, and if each possible decision value is a decision value of some reachable configuration [48]. A process is non-faulty provided that it takes infinitely many steps during a run, and is faulty otherwise [48]. A run is admissible provided that all messages sent to non-faulty processes are eventually received, and that at most one process is faulty [48]. A run is a deciding run provided that at least some process reaches a decision in that run [48]. Lastly, a consensus protocol is totally correct in spite of one fault if it is partially correct and every admissible run is a deciding run [48].

Now, lets assume an asynchronous consensus protocol P that is totally correct in spite of one fault. Then by the first lemma there exists a bivalent configuration C_0 for P. Now, according to the second lemma we can reach a bivalent configuration C_1 by delaying the first applicable message to C_0. Similarly, we can reach another bivalent configuration C_2 from C_1. And C_3 from C_2. Indeed, we can continue constructing this admissible but non-deciding run forever and so it follows that P is not totally correct in spite of one fault.

Theorem 1. *The FLP result: No consensus protocol is totally correct in spite of one fault [48].*

It is sometimes claimed that FLP result is not applicable to real world because it is proved on an unrealistic system model: the scenarios used to prove it never occur in real world [55]. It is certainly true, that the FLP result does not hold in synchronous systems (e.g. a LAN with a 30 second timeout to detect crashed processes), but there is a trade-off between reducing the probability of incorrect failure suspicions, and fast reaction to process failures, a path which quickly leads back to the domain of the FLP result. For example, consider a consensus algorithm deployed on a system, where a 30 second timeout is used to detect failed processes. Since 30 seconds is such a long time we can be fairly certain on the ability of the system to detect failures correctly. Thus we can describe this system as adequately synchronous and

not worry about the FLP result. But 30 seconds is a long reaction time in case of a failed process, e.g. the failure of the leader would cause the whole system to become unavailable for at least these 30 seconds. In a time-critical application the natural thing to do would be to reduce the timeout interval, but this would then increase the probability of incorrect failure suspicions! If we continue this path, we will soon cross the point where the probability of incorrect failure suspicions becomes non-negligible, and where the system becomes asynchronous. And as soon as the system becomes adequately asynchronous the FLP result start to hold again [55].

But what are the implications of FLP result? Does this result prevent the implementation of fault-tolerant consensus protocols? Is it not the whole point of fault-tolerant consensus to arrive at a decision despite of failures? Yes, there is still life after the FLP result [55]. The FLP result simply states that consensus can not be achieved in every possible run, that there exists some runs that that no consensus protocol can handle [55].

The practical implications of the FLP result are simple. Algorithm designers should better characterise the prevailing conditions that are required for reaching a decision in their algorithms (e.g. is the system assumed to be partially synchronous, timed asynchronous, or asynchronous with failure detectors) [55]. Even more importantly, they should take into account that sometimes the system can not live up to these requirements, and thus clearly state if their algorithm favours liveness or safety in such situations [55]. Other practitioners should take the FLP result as a warning: relying on a consensus algorithm, no matter how fault-tolerant, has the slight possibility of having to give up either the liveness or safety of the system.

5.2 CAP

At The Symposium on Principles of Distributed Computing (PODC) 2000 Eric Brewer made his famous CAP conjecture: it is impossible for a distributed system to be simultaneously consistent, available, and partition tolerant [56]. Later, in 2002, CAP conjecture was formally proved by Gilbert and Lynch, and became widely known as the CAP theorem [56].

Theorem 2. *CAP: in a network subject to partitions (P), it is impossible for any distributed system, that maintains linearisable consistency (C), to be always available (A) [56].*

Since there exist many different levels of consistency and availability, and different

types of partitions, we must define what is really meant by each of these before going into the proof of the CAP theorem. First off, consistency in CAP is defined as linearisable [56]. Next, availability in CAP is a simple requirement that each request must eventually receive a response [57]. A fast response is of course preferable to a slow one, but in the context of the theorem, even a slow response is sufficient to cause trouble [57]. Finally, by partitions it is meant that the communication is unreliable: at any moment nodes can be partitioned into multiple groups that can not communicate to each other. During these arbitrarily long time periods messages can be delayed or sometimes lost forever [57].

Informally the proof of the CAP theorem is rather simple [56]. We illustrate the idea of the proof by the following scenario. Let us assume that there exists an algorithm A that is strongly consistent, available, and partition tolerant. Assume that the network consist of at least two replicas, so that it can be split into two disjoint, non-empty sets $\{G_1, G_2\}$. Next, assume that a partition occurs in such a way that all messages between these two sets are lost. It follows that replicas in both G_1 and in G_2 are unaware of any writes that may occur in the other set for a particular key k. Now, further assume that a client requests to read the value of k from a node g_2 in G_2. What value should the node respond with? It is possible that no writes have been made to replicas in G_1 after the partition, and the value of k is still the last writes result v_1. But it is equally possible that another such a write has occurred after the partition changing the value of k to some v_2, where $v_1 \neq v_2$. Since these scenarios are indistinguishable from the perspective of g_2, it can not determine whether to return value v_1 or v_2. Hence, A has to choose between availability (possible returning a stale value) and consistency (to wait indefinitely), a scenario which contradicts our first assumption that there exists such an algorithm A, that can remain strongly consistent and available despite of partitions.

The CAP theorem is an useful tool to have at hand. According to this theorem most systems fell into either AP or CP category (during a partition they either give up liveness or safety). It enables us to quickly categorise systems we deal with in the real world and if they are suitable to our needs (by the CAP theorem we are usually looking for a system that is either CP or AP).

As a quick example, let us see how Dynamo [10], Amazon's always writeable key-value store [10], gets categorized according to the CAP theorem. Dynamo uses a quorum based consistency protocol, sloppy quorum, which enables nodes to store writes on behalf of unavailable ones. Therefore Dynamo is able to continue to stay

available even though the client's designated home replicas have become unavailable. This naturally comes with a cost in consistency, and thus replicas are allowed to diverge, a situation which is reconciled with occasional read repairs when needed [10]. So it comes with no surprise that Dynamo can be categorised as an AP system.

As an another example, let us next categorise PNUTS, Yahoo's distributed relational database. Since eventual consistency is too weak for Yahoo's web services, PNUTS was designed to be per record sequentially consistent [33]. This means that PNUTS has to give up accepting either writes or reads during a partition (see subsubsection 4.4.3). Since PNUTS is used mainly to serve content for Yahoo's web services, it makes more sense for PNUTS to favour reads over writes, and thus writes are restricted during major outages [33]. Therefore PNUTS reduces availability (for writes) during partitions is thus a CP system.

The CAP theorem has also received its fair share of criticism over the years [58, 59]. Particularly, the original "2 out of 3" formulation of the theorem lead to much confusion, and has been since deemed as misleading [60]. Firstly, the CAP theorem prohibits just a tiny part of the design space: only linearisable consistency paired with perfect availability is not possible in the presence of network partitions (which are rare) [60]. Secondly, the choice between consistency and availability does not have to be system wide, and instead can be done individually for each subsystem [57]. Finally, all of the three properties C, A, and P are more continuous than binary: system can be available for a certain percentage of the time, consistency comes in many levels, and even partitions differ in their severity [60].

5.3 PACELC

Given that early distributed systems design was keenly focused on strong consistency, it is natural to assume the CAP theorem as the major reason many distributed systems architects started experimenting on weaker consistency models [59]. The reasoning behind this assumption was that since every distributed system is build upon an unreliable network they have to be designed with either reduced consistency or availability [59]. But the CAP theorem applies only to a fraction of the operation of a system; that is, when there is an actual partition in the system. Thus, baseline operation of a system should not be affected by the CAP theorem, which makes the assumption, that the CAP theorem was the reason behind a surge in weaker consistency systems, flawed. But if the trend in weaker consistency models was not motivated by the CAP theorem, then by what?

It turns out that latency is a critical factor in online transactions: even an increase as small as 100 ms can dramatically reduce the probability that a customer will continue to interact with the service or to return in the future [61]. Keeping this in mind and considering that many of the important post CAP theorem distributed systems, such as Dynamo, Cassandra [62], and PNUTS are designed as databases for web services, it is no wonder that these systems aim for extremely low latencies [59].

But unfortunately, there is a fundamental trade-off between consistency and latency [59]. As a side note, latency is arguably same as availability since an unavailable system is essentially a system with extremely high latency, and thus a system becomes more available as its latency reduces [59]. But let us return back to the trade-off. As we know by now, one of the reasons to replicate data is to avoid unavailability caused by failing nodes. Therefore, in order to remain available systems are required to replicate writes before anything bad happens to the node that accepted them. And as we already know, this replication is done according to the consistency model of the system, which stipulates requirements such as the number of followers needed to accept a write before it can be safely acknowledged. These requirements get stricter as the consistency model gets stronger, causing the system to become more latent in acknowledging writes (less available). Thus, tightening consistency requirements increases the latency experienced at the client [59]. This trade-off between consistency and latency persists even when the system is under normal operation and is thus separate from the trade-off presented by the CAP theorem.

Thus, many of the new distributed systems were designed with lowered consistency requirements mainly to reduce their response times, not because of the CAP theorem [59]. Consequently, a more complete portrayal of the space of potential trade-offs is required; one that takes the trade-off between consistency and latency into account.

Definition 5. PACELC: if there is a partition (P), how does the system trade off availability and consistency (A and C); else (E), when the system is under normal operation, how does the system trade off latency (L) and consistency (C) [59]?

To become more familiar with this useful tool, lets analyse four different distributed systems from each trade-off category of PACELC: PA/EL, PC/EC, PA/EC, and PC/EL.

Dynamo is a PA/EL system [59]. We already found out, that according to the CAP theorem, Dynamo is a PA system since it reduces consistency from quorums to

sloppy quorums during partitions. But even though quorum is a stronger consistency guarantee than sloppy quorum, it is still a rather weak consistency model because it allows inconsistencies to arise (a write reaching the quorum but not all of the replicas). Dynamo thus risks occasional inconsistencies in order to avoid waiting on staggering replicas, which makes it a PA/EL system [59].

BigTable [31] is a PC/EC system [59]. In BigTable, tablets are assigned to a single tablet server at a time [31]. BigTable is thus unable to give up consistency to become more available or less latent, which makes it a PC/EC system.

MongoDB [63] can be classified as a PA/EC system [59]. Under normal operation it guarantees reads and writes to be consistent [59]. However, if the master node gets partitioned from the rest of the system, it stores all writes it has received but not yet replicated in a local rollback directory [59]. Meanwhile, the rest of the system elects a new leader to remain available [59]. Thus, the new master and the old master become inconsistent until the rollback directory of the old master becomes available again and is applied against the new master [59]. Hence, according to PACELC, MongoDb can be classified as a PA/EC system since a partition causes more consistency than availability issues [59].

PNUTS is a PC/EL system [59]. We already analysed PNUTS with the CAP theorem and concluded it to be a PC system since it does not lower its sequential consistency requirement during partitions. However, under normal operation PNUTS favours availability over consistency, making it a PC/EL system [33].

5.4 Harvest and Yield

In practice many systems reduce consistency or availability instead of choosing one entirely over the other [30]. Such systems remain available through graceful degradation of functionality. This degradation can be characterised as a trade-off between harvest and yield: in the presence of a fault there is typically a choice between giving an imperfect response (reducing harvest) and providing no answer (reducing yield) [30].

Yield is the fraction of queries that are completed [64]. It is an practical availability metric that all practitioners should be familiar with. But even though yield measures availability it should not be confused with the availability of CAP and PACELC. Rather than giving the probability of a response during a fault, yield gives the long-term probability of a response [30]. Yield is thus numerically very close to uptime,

but is more useful in practice since it directly maps to user experience [64]. Not all downtime affects the user: being down for a couple of seconds during off-peak and peak times gives the same uptime, but vastly different yields, because there might be a many orders of magnitude difference in load [64].

Definition 6. Yield = queries completed/queries offered [64],

Harvest is the fraction of data that is reflected in a response [30]. For example, assume a distributed database of 100 nodes where individual node faults are tolerated. Further assume that each of these nodes is assigned with a single shard without replication. Therefore removing a node removes a proportional fraction of the available data. Now consider a search engine that searches this database: as nodes become unavailable the search results get increasingly inaccurate as the available data diminishes. Thus the yield stays the same, but harvest is reduced.

Definition 7. Harvest = available data/complete data [64].

To understand how harvest and yield apply to writes, consider an AP system that is partitioned into two sets of nodes A and B. Since the system is an AP system each of the sets can continue to stay available for their respective clients. But updates sent to the system only affect the reachable set and remain unseen on the other side of the partition. This is a form of reduced harvest: the available audience has decreased. Now assume that a client of A wants to update some data that is only reachable in B. Since there is no way for nodes in A to handle this request the only solution left is to let the request fail (to reduce yield).

5.5 Delay-Sensitivity Framework

Availability of a service can be defined as the proportion of requests that meet some latency bound (e.g. as described by the SLA of the service) [58]. With this alternative definition for availability, we can measure the tolerance of a service to network problems by analysing how its operation latency is affected by changes in the network delay, and whether it can stay available according to its SLA [58]. This method is called the delay-sensitivity framework and provides tools for reasoning about trade-offs between consistency and robustness to network faults [58].

To replace CAP with this latency-centric viewpoint we need to examine how operation latencies are affected at different levels of consistency [58]. Fortunately, there

already exists several impossibility results providing lower bounds on the operation latency as a function of network delay [58]. These results show that any algorithm providing a particular consistency model can not perform better than some lower bound [58].

Any algorithm relying on a linearisable read-write register with at least two readers and one distinct writer, must have an operation latency of at least $u/2$ for both reads and writes, where u is the uncertainty of delay, which can be as high as the network delay d [42, 65]. In other words, linearisability requires operation latencies of both reads and writes to be proportional to the network delay d [58]. Thus, under linearisability read and write operations have a latency of $\mathcal{O}(d)$ making them delay-sensitive, as their latency changes proportionally to the network delay [58].

Any algorithm relying on a sequentially consistent read-write register must have $r + w \geq d$, where r is the latency of a read operation, w the latency of a write operation, and d the network delay [66]. Interestingly this result grants us some degree of freedom when designing sequentially consistent systems. For example, we can reduce the average operation latency of a write-heavy application by choosing $w = 0$ and $r \geq d$ (and vice versa for read-heavy applications) [58]. To summarise, sequential consistency allows either reads or writes to be delay-independent ($\mathcal{O}(1)$) but requires the other class of operations to remain delay-sensitive ($\mathcal{O}(d)$) [58].

But what is the strongest consistency model that enables all operations to be delay-insensitive ($o(1)$)? According to recent studies causal consistency meets this description [67, 68]. It follows that all of the weaker consistency models are also delay-insensitive [58]. Nevertheless, this result does not make them obsolete. Better performance is still a valid reason to pick a weaker consistency model [45]. But if delay-insensitivity is the only requirement then causal consistency is the optimal solution [58].

Since network delays vary between different nodes, we have to choose carefully which delay to use for $\mathcal{O}(d)$ when analysing a system with delay-sensitivity framework [58]. It is best to start by analysing the communication patterns of the system at hand against its network topology. A very effective strategy is to split delay d into intra data centre delay d_{local} and inter data centre delay d_{remote}, where $d_{local} \ll d_{remote}$. This is why some systems have arrived to designs where different consistency models are used for different parts of the system. For example, in Cluster of Order-Preserving Servers (COPS), a distributed key-value store, every operation is linearisable within a data centre ($\mathcal{O}(d_{local})$), but the effects of these operations are replicated

with causal consistency across data centres ($\mathcal{O}(1)$) [32]. Thus, even though COPS contains delay-sensitive operations, none of them are delay-sensitive towards d_{remote}. This way clients of each data centre can enjoy linearisability with relatively low operation latencies while COPS can stay highly scalable in respect to the number and locations of its data centres [32].

It should be noted that although delay-insensitive algorithms are decoupled from the network delay, replication still takes time proportionally to the delay [58]. In other words, even though a delay-insensitive request is able to return in $\mathcal{O}(1)$ time, it still takes $\mathcal{O}(d)$ for its results to become visible [58].

6 Avoiding Coordination

Minimising coordination is the key to maximising scalability, availability, and high performance in distributed systems [69]. Coordination, the requirement for concurrent operations to communicate synchronously or otherwise wait for each other in order to complete, is expensive [69]. But coordination-free execution is not always safe: it might compromise application level correctness, or consistency [69]. For example, in a banking application, concurrent and coordination-free withdrawals can cause an account balance to become negative [69]. To prevent such undesirable outcomes, the application must coordinate the execution of these operations [69]. Hence, in order to maximise the coordination avoidance of a system, we first need know which parts of the system can safely work without coordination.

In subsection 6.1, we start with a discussion on consistency as logical monotonicity (CALM) principle, a method used to reason about when distributed code can be executed safely without coordination. Next, we move onto subsection 6.2 to discuss commutative replicated data type (CRDT) and examine how by requiring concurrent operations to commute we can ensure eventual convergence and thus avoid conflicts. Lastly, in subsection 6.3 we finish with a discussion on invariant confluence (I-confluence), a framework used to determine whether an application requires coordination for correct execution.

6.1 CALM Principle

The CALM principle is a technique to help distributed systems programmers to reason about the consistent behaviour of their code in the face of temporal non-determinism, including the reordering and delay of messages and data across nodes [70]. It answers questions such as: Where in distributed system is eventual consistency good enough? How can we be sure that these eventually consistent components do not taint other parts of the software? How can we maintain such code?

A program can be guaranteed eventually consistent if its execution is independent of any temporal non-determinism [70]. We call such programs order independent [70].

Monotonic programs—e.g. programs expressible via selection, projection, and join—are order independent [70]. In such programs the final order of the input will never cause any earlier output to be revoked [70]. Thus they can be implemented by streaming algorithms that incrementally produce output as they receive input [70].

In other words, in a monotonic program, any true statement continues to be true as new axioms, including new facts, are entered into the program [70]. On the other hand, non-monotonic programs, e.g. programs that contain operators such as negate or aggregate, always require some degree of coordination since they have to inspect all of the input before any output can be produced [70].

Monotonic programs are easy to distribute: they can be implemented through streaming algorithms that produce actionable outputs to consumers while tolerating message delay and reordering from producers [70]. In contrast, even simple non-monotonic programs are difficult to get right in distributed systems [70].

As an example, consider a program $f(X, v)$ which purpose is to find out if the minimum of set X is below value v ($\texttt{MIN}(X) < v$). Let us assume that the set X is so large that it can not be processed by f as whole. Thus, we have to split X into subsets $X_1, X_2, \ldots, X_n \subset X$ which we then input into f separately. To save time we distribute the calls to $f(X_i, v)$ which gives arise to the question: should the invocations of f be coordinated or not? To answer this question we need to find out if f is monotonic. By virtue of the semantics of `MIN` and $<$: once a subset X_i satisfies $\texttt{MIN}(X_i) < v$, any superset of X_i, e.g. X, will also satisfy it [70]. This means that f is indeed monotonic and thus X can be safely processed without coordination as follows: $f(X, v) = f(X_1, v) \vee f(X_2, v) \vee \ldots \vee f(X_n, v)$.

This brings us to the crux of the CALM principle: the tight relationship between Consistency and Logical Monotonicity [70]. Monotonic programs guarantee eventual consistency even when faced with temporal non-determinism [70].

We can use CALM principle to safely minimise coordination in distributed programs by only coordinating the points of non-monotonicity [70]. A simple syntactic check is a good start: if the program only contains monotonic operators it is monotonic and can be implemented without coordination, regardless of any read-write dependencies [70]. On the other hand, if non-monotonic symbols are found they may require coordination to ensure consistency and should be treated accordingly [70].

6.2 CRDTs

The CRDTs are a family of data structures on which all concurrent operations commute [71]. By ensuring non-concurrent operations to be delivered in causal order and concurrent operations to commute, CRDTs are guaranteed to eventually converge and never conflict [72]. Thus, they require no coordination [71]. As a

result CRDTs can remain available and scalable even during high network latency and partitions [72].

As an example, consider a website which hosts pictures that users can like or dislike. The pictures are stored in a central repository together with their likes. Now let us assume that by either liking or disliking a picture the client simply adds or subtracts one from the local number of likes for a picture and sends this updated number to the central repository. This approach is certainly simple but unfortunately is not commutative: whenever a user likes or dislikes a picture the new number of likes will disregard all possible likes and dislikes which might have taken place while the user was contemplating whether to like the picture or not! In an implementation such as this some kind of coordination among the clients would be required to prevent likes or dislikes from becoming lost. Let us now consider another approach that is commutative: instead of sending the updated number of likes to the central repository, the client first updates the local number of likes and then proceeds to send either plus (like) or minus (dislike) one to the central repository which then adds the received plus or minus one to the number of likes stored at the repository. Now, since both likes and dislikes commute we can conclude the pictures in this system to be CRDTs and thus operations on them require no coordination to converge. To receive the most up-to-date number of likes for a picture the client simply asks for the central repository for this number.

6.3 Invariant Confluence

I-confluence is an another framework that can be used to determine whether an application requires coordination for correct execution [69]. This is achieved by enabling application developers to specify their correctness criteria in the form of invariants [69]. After all, since the underlying distributed system has no idea what the application considers as consistent, so why not let the application developer decide?

Before going into I-confluence, we must first define what is meant by invariant valid (I-valid) replica state and $I-T$ reachable state. Replica state R is I-valid if and only if $I(R) = true$, where I is an invariant specified by the application developer [69]. Moreover, we say that a system is globally I-valid if and only if all of its replicas are always I-valid [69]. An $I-T$ reachable state is a state that can be reached with an invariant I, a set of transactions T, and a merge function in such a way, that each intermediate state produced by transaction execution or merge invocation is

also I-valid [69].

Now that we have defined I-validity and $I-T$ reachable state, we are ready to define I-confluence. A set of transactions is I-confluent, with respect to invariant I if for all $I-T$ reachable states D_i and D_j, with a common ancestor state, the merging of D_i and D_j is I-valid [69]. To put it more simply, I-confluent transactions will never lead to an invalid state (against invariant I) in any of the replicas, regardless of how they are propagated in the system.

It can be shown that a globally I-valid system can execute a set of transactions T with availability, convergence, and coordination-freedom if and only if T is I-confluent with respect to invariant I [69]. I-confluence is thus a necessary and sufficient condition for coordination-free, invariant preserving execution [69]. In other words, if I-confluence holds there exists a coordination-free, correct execution path for the transactions; if not, there can be no implementation that guarantees these properties for the provided invariants and transactions [69].

But I-confluence is not a silver bullet. Firstly, simply because an application is I-confluent, it does not indicate that all of its implementations perform equally well [69]. I-confluence only guarantees that a coordination-free implementation exists [69]. Secondly, I-confluence can only guard against violations of invariants that are provided [69]. Developers are thus required to either guarantee the correctness and completeness of their invariants, or to opt for more conservative analysis or mechanism, such as employing serialisable transactions [69]. In practice, the first option is seldom feasible.

7 Building a Maintainable Distributed System

Building a distributed system is always a complicated undertaking and the danger of ending up with an unmaintainable system is real. Thus, it is of utmost importance to keep the system as simple and operable as possible to minimise future costs.

In subsection 7.1, we start by a discussion on importance of service-level monitoring. Next, in subsection 7.2 we discuss logs and the importance of log aggregation and other logging practices. Then, in subsection 7.3 we examine different distributed systems debugging approaches. Next, in subsection 7.4 we discuss how platform-level monitoring complements service-level metrics. Then, in subsection 7.5 we explain how back-pressure can help mitigating the risk of cascading failure. Next, in subsection 7.6 we discuss containers and their various benefits. Then, in subsection 7.7 we examine how handling of containers can be automated with container orchestration platforms. Lastly, in subsection 7.8 we finish with a discussion on differences between centralised and decentralised distributed system architectures.

7.1 Service-Level Monitoring

Service-level monitoring informs operators on how their system is changing over time. Analysing long-term trends reveals important aspects like how fast a database is growing or if the system really is slower than it was the previous week. Knowing how the behaviour of the system is different from the behaviour preceding a component change illustrates the difference between successful engineering and failed shamanism [9].

Service-level monitoring alerts operators when something breaks, but alerts can be hard to get right. For example, alerting too eagerly can cause operators to dismiss real situations as false positives, whereas alerting only in extreme situations usually means alerting too late [8, Chapter 2]. It can be tricky to find the golden middle way.

Service-level monitoring enables operators to set up dashboards. A good dashboard is simple and informative: a quick glance should be enough to tell the operators if the system is performing as expected. An informative dashboard can help operators to discover problems even before the alerts begin to arise.

7.2 Logging

While service-level monitoring informs operators when something is wrong, logging informs operators what is wrong. However, logs tend to get filled with all sorts of odd bits and bobs [9]. Thus, it is important not to over-emphasize something seen in the log before its importance is checked against the monitoring metrics [9].

Since logs in distributed systems are product of many services, they tend to get spread all over the system and in different formats. Hence to avoid wasting operators' efforts logs should be aggregated to a central location in unified format for easier access and better intelligibility. Log aggregation also opens doors to other good practices such as log rotation, analysis, search, and easier long term log storage.

7.3 Debugging Distributed Systems

Although service-level monitoring and logging helps operators to see when and what is wrong in the system, they usually lack the required information to indicate why something is wrong in the system [8, Chapter 2]. Consequently, we need tools to understand the complex interactions between many programs, possibly running on hundreds of servers [8, Chapter 2]. Distributed system tracing tools have been proposed to fulfil this need [8, Chapter 2].

Black-box monitoring systems, such as WAP5 [73], treat the system as a collection of black-boxes. They monitor the traffic within the system and use statistical methods to infer causal relations [8, Chapter 2]. This approach has the advantage of not requiring any assistance from software infrastructure, but this advantage comes with a cost in information accuracy [8, Chapter 2].

For better accuracy, instrumentation-based tracing schemes, such as Dapper [74], Pip [75], and X-trace [76], are recommended [8, Chapter 2]. But these tools have the downside of requiring all components of the system to be instrumented to collect comprehensive data [8, Chapter 2]. More specifically, instrumentation-based tracing relies on every record being explicitly tagged with a global identifier that links it to the original request [74].

7.4 Platform-Level Monitoring

Before we can begin to properly understand and analyse service-level metrics, we must monitor the computing platform as well. Although the misbehaviour of a hardware component can sometimes be inferred from service-level metrics, it is still an indirect assessment [8, Chapter 2]. Furthermore, since distributed systems are often designed to tolerate hardware-faults directly in the software, monitoring at these levels can cause a vast number of underlying hardware problems to go unnoticed, allowing them to build up until they can no longer be mitigated [8, Chapter 2]. At that point the following disruption could be severe. Hence, in order to really understand what is going on in the system, tools that continuously and directly monitor the health of the computing platform are required in addition to the service level metrics [8, Chapter 2].

7.5 Back-Pressure

Back-pressure is a feedback mechanism that enables systems to gracefully respond to load rather than to collapse under it. It is about the signalling of failure from a serving component to the requesting component and about how the requesting component handles those signals to prevent them both from overloading [9]. This may involve dropping new messages, or shipping errors back to the requester [9]. Timeouts and exponential backoffs on connections to others systems are also important features of back-pressure [9].

Without back-pressure, cascading failure becomes likely: a service that is not prepared for the failure of another, tends to emit failures to other services depending on it [9]. Moreover, the feedback provided by back-pressure is a very informative metric to monitor.

7.6 Containers

In the recent years, the software industry has seen a dramatic rise in adoption of container technology [77]. Containers are a packaging mechanism, that abstracts applications from the environment they actually run in [78]. This abstraction enables containerised applications to be deployed consistently, regardless of the target environment.

Containers are often compared with virtual machines. Virtual machine is essentially

a guest operating system that runs on top of the host operating system with virtualised access to the underlying hardware [78]. Similarly to virtual machines, we use containers to package applications together with their dependencies into isolated environments. But instead of virtualising the hardware stack, containers virtualise at the operating system level and run directly on top of the kernel [78]. Consequently, containers are much lighter: they share the kernel, and most importantly they start much faster and require only a fraction of the memory since they do not need to boot an entire operating system [79].

Containers enable developers to create predictable environments that are isolated from other applications and include all dependencies required for an application to operate [78]. Moreover, the environment is guaranteed to be consistent no matter where the container is ultimately deployed. This means that the developers can spend less time debugging differences in environments, and more time shipping new features for users.

Containers provide a higher level abstraction to process life cycle management [80]. Every container effectively exports three functions: start, stop, and pause [77]. Although the interface is extremely limited, we can provide a finer interface by programming the container to host a web server at specific endpoints [77]. To the outside direction we can expose application information, such as monitoring metrics and logs [77]. And to the inside direction we can expose an interface for finer life cycle control. Most importantly, this finer life cycle management interface enables containers to be orchestrated across multiple nodes.

7.7 Container Orchestration

Container orchestration platforms are frameworks for integrating, deploying and managing containers at scale. They work by having each node host an agent that manages containers locally and advertises resources to the master node. The master pools the advertised resources and uses this information to schedule deployments. Besides where and when to deploy an application, the responsibilities of the master include load rebalancing and exposing an interface for operators to monitor and deploy their applications.

Container orchestration platforms effectively bind distributed hardware resources into a single pool of resources [78]. With container orchestration in place, operators no longer need to concern themselves with choosing a node to deploy their applica-

tion on. All that is required is to describe how the application should be deployed and managed afterwards. The orchestration platform then ensures that the state of the application stays as close as possible to the desired state by continuously changing the state of the application towards its deployment description.

Table 1 is a comparison of some of the most popular container orchestration platforms. From the compared platforms, Kubernetes and Docker Swarm Mode are pure container orchestrator frameworks, while Nomad and DC/OS provide other features as well.

7.8 Architecture

There are basically two types of distributed system architectures: centralised and decentralised. Centralised architectures are coordinated by a single dedicated coordinator, often called as the master. Alternatively, in decentralised architectures the coordination logic is spread among the system in such a way that the components of the system are able to make decisions together or autonomously.

The greatest advantage of centralised architectures is their simplicity. Since there is only one coordinator, it can have global knowledge on the state of the system and thus make sophisticated decisions [11]. For example, GFS is coordinated by a centralised master [11]. The master maintains a table containing mappings from files to chunks and locations of each of these chunks and their replicas [11]. The master populates this tablet at startup by polling the chunk servers for the chunks they are holding. As the master makes all the chunk placement decisions it is relatively simple to keep this table up-to-date: all that is required is to monitor the chunk servers with heartbeat messages and modifying the tablet accordingly [11]. Because of this global knowledge, the master can make sophisticated decisions on aspects like which chunks to re-replicate, where to create new ones, and if rebalancing is required [11].

As the slowness or failure of the master can cause the whole system to stall, it is important to avoid involving the master in too intensive or too many operations [11, 31]. For example, GFS avoids overwhelming the master by limiting its involvement in data transfers [11]. When a client wants to read or write to a chunk, it only contacts the master if it does not already know the location of the chunk. Client caches this information and sends all subsequent requests directly to the leader replica of the chunk, which in turn replicates all the changes to the follower chunks

without contacting the master.

Even though centralised architecture introduces a single point of failure into the system, the risk stays relatively low. After all, even though distributed systems are often experiencing faults of some sort, the individual nodes are still very reliable. The problems come from the sheer number of machines: the mean time between failures (MTBF) of some component is always due. Thus, although faults are common, the failure of the master remains unlikely [81].

By comparison, decentralised architectures do not have to design around bottlenecking the master, but need additional protocols for agreeing on the state of the system and for avoiding conflicts. These protocols mixed with the distributed coordination logic can make decentralised architectures complex and hard to reason about.

8 Current Production System

In this section the shortcomings of the current production system are laid out. We start by describing the system in sufficient detail in subsection 8.1 and end with and examination of its various problems in subsection 8.2.

8.1 Description of the Current System

The subject production system is a framework for turning raw telemetry data, weather predictions, and weather models into other weather predictions and models. This framework is required to abstract away the actual production from its description. This way the different production processes become easily monitored and configured through the production system.

The current production system consist of roughly 100 servers connected via a high-speed LAN. The system has about 10000 tasks, around half of which are launched, monitored, and managed by ecFlow [82], and the rest by a similar in-house solution. These tasks access and store all their data through a central on-premises NFS cluster. We monitor the infrastructure with Nagios [83].

EcFlow is centralised work flow framework that can run large numbers of jobs with various triggers. In a nutshell ecFlow works as follows: the master (ecFlow server) periodically checks if tasks should trigger and instructs the nodes holding the tasks to execute them.

The task trigger information is passed to the ecFlow server by a suite definition file. EcFlow server scans this file every minute and launches all tasks which trigger. A trigger can be basically anything, but most often we use checks to see if some relevant data has changed or if some other task has finished. Tasks that need to be run in specific dates or intervals can be made into cron jobs.

The suite definition includes the tasks' triggers but not the tasks themselves. Instead, the tasks are defined in ecf scripts which are basically shell scripts with additional work flow management commands. The locations of each ecf script is stored in the suite definition. Upon triggering a task, ecFlow server instructs its home node to execute it.

The in-house solution is very similar to the ecFlow framework but relies entirely on cron, ssh, and shell scripts. Instead of a suite definition the tasks are located in a single folder that gets polled every minute by a cron job. Each of the tasks in

this folder is named after the data that triggers it. Specifically, the filename of a task is the pathname to the folder where its input data is stored. Every time the cron job executes it scans the folder containing the tasks, parses the task names into pathnames, checks if the folders pointed by these pathnames contain new data, and executes tasks accordingly. These tasks are ordinary shell scripts containing remote instructions to workers (chosen when the tasks were written). All tasks are retried N times, but if they remain unsuccessful a notification is made via Nagios.

We store all data in a central on-premises NFS cluster. The data is organised by its source and type into a hierarchical folder structure, where each data source has its own folder. Whenever new data is produced (or arrives), it is timestamped and stored at some designated folder of the data source.

We have several monitoring systems installed for different metrics. The ecFlow half of the production system is exposed though the graphical user interface (GUI) client of ecFlow, which shows the state of all ecFlow tasks with one minute granularity. Unfortunately, this kind of task tracking is completely lacking from the in-house solution. In addition to the GUI, we monitor the traffic at the outward facing servers with pingdom [84], the health of the computing-platform with Nagios, and the performance of applications with New Relic [85].

8.2 Shortcomings of the Current System

The greatest problem with the current production system is how the task are assigned statically. Thus, load balancing relies solely on the operators' expertise to select the workers to handle new tasks. This choice is especially hard since it is practically impossible to know beforehand the load at different workers at the time the new task is executed. An unfortunate guess can lead to the task being assigned to a node that is maximised out every time the task is triggered.

An another problem with the system is that it lacks redundancy. Neither the ecFlow framework nor the in-house system is flexible enough to allow for task replication. Consequently, whenever a worker node becomes unavailable, its tasks are not executed until it recovers.

In addition, the I/O throughput of the on-premise NFS cluster is slowly turning into a bottleneck. Until now, we have managed to scale the system by simply adding more worker nodes. This approach has so far worked well and given us linear scaling since the workers work in isolation. But despite this isolation the workers still have

to compete on the I/O of the NFS cluster. Consequently, we can no longer expect linear scaling in the future.

The I/O throughput is not the only problem with the shared NFS cluster. The whole approach of using a shared file system as a object store is somewhat problematic. Firstly, in file systems the data access can only be controlled through permission flags. This means that in the system a misconfigured task might corrupt the data another if we are not careful. Secondly, the way we access data directly from the file system makes the tasks highly coupled to specific paths on the file system. Hence, if we want to relocate data we are forced to change the related tasks as well. Thirdly, since we directly use file system, the only way for us to locate and organise data is by pathname. This forces us to use clumsy timestamping practices to prevent new data from overwriting old data: each data is timestamped by appending its creation time to its pathname. Unfortunately, since some data sources are not governed by us, these data sources might format their timestamps differently, and thus the only general way to find the most recent data of each data source is to completely disregard the timestamps, and instead ask the file system for the last modified file in the folder. Fourthly, file systems lack hooking capabilities to inform external systems when new data has arrived. Consequently, we have resorted to poll the data source folders for changes, which certainly does not help with the limited I/O capacity of the NFS cluster.

The current production system moves data to the tasks rather than the other way around. So far this has been sufficient since most of the data is produced in-house and made directly available in the on-premise NFS cluster. And until now, most of the data produced outside has been either sufficiently small or produced so infrequently, that downloading it to the NFS cluster has not been a problem. But the recent developments in open data movement have made available a plethora of new data, some of which are too large or too frequently produced to be downloaded here.

9 New Production System

We start laying out the requirements for the new production system in subsection 9.1. Next, we design the new production system according to its requirements in subsection 9.2. Lastly, we finish by presenting a practical implementation of this design in subsection 9.3. We do not claim that the presented solution is optimal. Rather, the presented solution is simply one design we arrived at by using the knowledge gathered from the research done for this study.

9.1 Requirements for the new System

This subsection contains the requirements for the new production system. These requirements are mostly based on our experience with the current system. We have also listed the non-requirements of our system: features usually required of similar systems but not required of the system at hand.

9.1.1 Requirements From the Shortcomings of the Current System

The following is a list of requirements based on the shortcomings of the current system. By fulfilling these requirements, the new production system should avoid the shortcomings of the current system.

1. To avoid the complexity of statical load balancing, the new system must have dynamic load balancing.
2. To avoid the problem of central NFS cluster becoming a bottleneck, the new database solution for the new system must tolerate greater read/write traffic.
3. To avoid large files, the new database must split large files into more manageable file chunks.

9.1.2 Requirements From Experience With the Current System

The following is a list of requirements based on our experience with the current system.

1. The production system should be split into three separate components: database, task pool, and coordinator.

2. Most of the tasks in the system are only interested in the newest data, indicating that the new database should be distributed evenly to avoid hot spots from forming.

9.1.3 Non-Requirements for the new System

The following is a list non-requirements for the new production system: features usually required of similar systems but not required of the new production system.

1. It is OK for the system to stop production for short periods of time since there are no dependent real-time systems.
2. Almost all files in the system are 'write once read many' by nature indicating that write conflicts in our new database are going to be a rarity.
3. New tasks are added into the system on a weekly basis indicating that the write traffic to the task pool is going to be very light.

9.2 Design for the new System

In this subsection we apply the distributed systems theory to design the new production system according to its requirements.

9.2.1 Designing Distributed Systems

This subsection presents the result of our efforts in converting distributed systems theory into practice. Although most of the theory is straightforward, there is so much of it and it so intertwined that we had hard time trying to decide where to begin with it. The following is an overview of what is involved in designing distributed systems in general.

Before starting to design any distributed system we must first know what is theoretically possible: to this end we use the impossibility results and trade-offs discussed in section 5. The FLP impossibility result, although important for distributed systems theoreticians, does not have much effect to designing distributed systems in practice. The CAP theorem shows that under network partitions we have to give up either availability or strong consistency. PACELC extends this result by pointing out that under normal operation we can have either low latency responses or

strong consistency, but not both. Harvest and yield points out that if we design the system to gracefully degrade under partial failures, the choice between availability and consistency does not have to mutually exclusive. Finally, the delay-sensitivity framework shows how stronger consistency models are more sensitive to changes in network delay than weaker ones.

Almost every choice in distributed systems revolves around availability and performance, and thus we should carefully analyse these requirements before considering anything else. We are especially interested in the lower boundaries, since these directly indicate how much there is leeway to trade-off for other nice-to-have properties.

An especially nice-to-have property in distributed systems is simplicity (see section 7), and since simplicity is greatly affected by the chosen consistency model (see subsection 4.4) choosing one should be our next step. Strong consistency models are generally easier to reason about and to work with than weaker models since they usually lead to less surprises like stale reads, lost writes, or write conflicts (see subsection 4.2). In other words, for the simplest possible system, we should choose the strongest consistency model that is able to conform to the set availability and performance requirements.

After consistency model has been chosen, deciding on a replication strategy should be easy. This boils down to choosing the number of followers and leaders (see subsection 4.1). More followers usually means better throughput and availability for reads (depending on the consistency model), while more leaders usually leads to better throughput and availability for writes (again depending on the consistency model). Nevertheless, too many followers will waste storage and computing resources and having more than one leader per shard can lead to write conflicts. Dealing with conflicts is a complicated business (see subsection 4.2) and thus single-leader replication should be carefully considered over its conflict prone alternatives. We could state the following: if the write traffic per shard is mostly consecutive then single-leader replication is often the optimal solution (see subsubsection 4.1.2).

Sharding introduces isolation which in turn translates to improved scalability and availability (see subsection 4.3). And by sharding large replicas into something more manageable we can improve performance. But most importantly: we can use sharding to reduce, or to completely prevent, conflicts by capturing the concurrent access patterns of each component. The overall effect should be that each shard ends up being large enough to avoid unnecessary overhead, but small enough to

prevent hot spots from forming. Lastly comes the decision on the actual sharding strategy. As a rule of thumb, hash range sharding should always be considered first because it alleviates the risk of hot spots, but if efficient transactions across ranges are needed then there is no way around key range sharding.

Lastly, if we can not find a sufficiently performant solution, we should look for coordination avoidance (see section 6). When eventual consistency is enough CRDTs should be considered since they guarantee conflict-free convergence. Moreover, if the invariants of the application are well known, I-confluence should be used to find out which operations are safe to be implemented without coordination. But if the application were to written from scratch, we recommend using the CALM principle to ensure eventual consistency. This can be easily achieved with Bloom [70], a programming language that has a built-in support for CALM checks.

9.2.2 Architecture

To keep the logical components of the system separately maintainable, we are going for a modular, service-based architecture, where each component should be separately deployable. Based on our experience (see subsubsection 9.1.2) the system should divided into three separate components: one for storing input and output data (to abstract away the shared file system), one for storing tasks (to enable dynamical load balancing as is required of the new system; see subsubsection 9.1.1), and one for coordinating the actual production. There are basically two possible ways to implement the dynamical load balancing: either by instructing the workers to fetch the tasks themselves or by instructing the coordinator to assign the tasks to the workers. However, since centralised architectures tend to produce simpler solutions than decentralised architectures (see subsection 7.8) and since 100% uptime is not required of our system (see subsubsection 9.1.3), the centralised solution to load balancing is chosen.

Here is how we picture the components of the new architecture to come together. Tasks and their related triggers are sent to the task pool via a client program. Task pool notifies the master whenever it receives a new task. Upon receiving such a notification, master downloads the advertised triggers and adds them to its in-memory metadata. Whenever the master fails, it repopulates this metadata by simply polling the task pool for trigger information upon recovery. Master then continues to periodically check its metadata for triggered tasks and distributes them to workers according to its best knowledge. To avoid overwhelming the master, we

wish to exclude it from all the heavy data transfers. Consequently, the master does not send the actual task to worker, but its identifier so that the worker can fetch it directly from the task pool. Upon receiving such an identifier, worker downloads the task, executes it, and informs the master of its result. All tasks access and store their data via a distributed object store.

9.2.3 Task Pool

The performance and availability requirements of read and write traffic are very disproportionate for task pool. According to the past usage of the current production system (see subsubsection 9.1.3) we assume that new tasks continue to be added roughly once a week. This means that the requirements for write traffic are basically non-existent. Contrastingly, the task pool is going to be read whenever a task triggers, which will happen frequently and often in batches. Thus, its read traffic requires high availability and performance.

Whenever a worker receives a task identifier it asks the task pool for the related task. We assume that the task can be found since it would not make any sense for an identifier to exist without the task. In terms of consistency, this translates to a requirement for writes to become immediately visible for all latter reads. When we combine this requirement with the earlier disproportionate availability and performance requirements, we can conclude that the strongest possible consistency model for task pool is sequential consistency tuned for fast reads (see subsubsection 4.4.3).

The task pool is frequently read, but rarely written into. Thus, the optimal replication strategy for the task pool is single-leader replication (see subsubsection 4.1.2). To account for the high availability and performance requirements of the read traffic, each leader should be accompanied with a sufficient number of followers. The exact number should be experimentally determined and scaled according to the read traffic.

The task pool is a distributed, persistent, three-dimensional unsorted map indexed by a task identifier and a column key. There are only two columns: one for the triggers and one for the actual tasks. The task pool is sharded into row ranges called tablets. These tablets should be large enough to avoid unnecessary overhead resulting from having to manage too many tablets, but small enough to be manageable by the weakest node in the system (see subsection 4.3). The exact size should be experimentally determined and tuned according to collected metrics. Since tasks

are never accessed by range, we can safely choose hash sharding as the sharding strategy and thus ensure best possible distribution of load across tablets.

9.2.4 Master

The master in the new production system is not replicated and hence there is not much to say about it in the sense of distributed systems. Since there is only one master its failure is going to be a relatively rare event and thus a simple failover strategy should be sufficient (a simple strategy should also suffice since 100% uptime is not required of the new system; see subsubsection 9.1.3). We recommend automatic server restarts in case of master operation failures, and if the server itself becomes faulty, manual migration to another server.

To enable master to continue where it left upon recovery, careful attention should be paid to persistent logging. Logs should include what triggered each task, the worker it was assigned to, and its result (if received). Such information enables master to avoid duplicate task executions. Upon recovery master downloads the triggers from task pool, checks for triggered tasks, playbacks the log, and launches only the tasks that are not being processed, whose triggers are more recent than trigger information in the logs, and failed tasks.

9.2.5 Distributed Object Store

The NFS cluster in the current production system is slowly turning into a bottleneck, signifying great data traffic (see subsubsection 9.1.1). To avoid the new solution from becoming a bottleneck as well, we should better utilise the current hardware. In a ideal production system all of the worker nodes should be continuously working on some task. This means that the new database should be always available for reads and writes. The data items in the system are mostly written just once (when created) but read many times (see subsubsection 9.1.3). Since data is barely ever modified, write conflicts on some single data item are going to be a rarity.

To support the strict performance and availability requirements of the new solution, we are going to need every drop of performance from the current hardware. This means that we can not compromise with any of the stronger consistency models. Thus, the strongest consistency model able to conform to these needs is eventual consistency.

Because of the high performance and availability requirements of the database, we

have to discard single-leader replication from the list of possible replication strategies, which leaves us with multi-leader and leaderless replication to choose from. Multi-leader replication with every node designated as a leader enables practically the same performance as leaderless replication. But leaderless replication can provide better availability if sloppy quorums are utilised (see subsubsection 4.1.4). Therefore, leaderless replication is chosen as the replication strategy for the new database. Furthermore, to meet the high availability, performance and durability requirements, (N, R, W) quorums should be initially configured to (3, 1, 2). We settled upon this configuration by starting from the recommended (3, 2, 2) configuration (see subsubsection 4.1.4), but reduced the read quorum to enable faster reads. Normally this change would increase the probability of returning a stale value, but since the data is mostly write-once we do not really have stale values.

We expect many of the data objects to be quite large; some exceeding tens of gigabytes in size (see subsubsection 9.1.1). Moving such objects around is slow and cumbersome. Therefore, we should split large objects into manageable size chunks before uploading them. This increases throughput since chunks can be uploaded and downloaded in parallel. In addition to increased throughput, storing chunks instead of objects makes many database maintenance operations such as rebalance, handoff, and anti-entropy faster. Operating on chunks also reduces the probability of forming hot spots since read traffic is automatically distributed among chunks.

We can prevent write conflicts by simply making each file chunk its own shard. This works because each file in the system always originates from a single source only and is usually never modified afterwards (see subsection 4.3). Thus, even though there are many concurrent sources, their writes will not conflict since they are isolated into different shards. But the data traffic is not limited to writes. Instead, every task has a set of input files, which are read upon its execution. Usually, the tasks are interested in only the most recent files in the system (see subsubsection 9.1.2). This means that the database is particularly susceptible to hot spots since read traffic is biased towards new data. Therefore, we should ensure that new file chunks are distributed as evenly as possible, meaning hash sharding. Hash sharding suits the database well since we already gave up on efficient key-range transactions when we decided to split large files into chunks.

Initially we thought of using CRDTs paired with dotted version vectors (see subsection 4.2) to ensure the convergence of the database, but we realised that a simpler approach was available due to the write-once nature of the data: timestamped writes

with LWW. We chose LWW as the conflict resolution strategy to ensure task idempotency: if for some reason a task gets executed twice, the second execution just overwrites the first one. This also applies to conflicts found during read repairs, hinted handoffs, and anti-entropy.

9.2.6 Maintainability

The majority of the costs of software come from its ongoing maintenance, not from its initial development. To minimise these maintenance costs we must make the new production system as maintainable as possible. We have already finished the most critical part in creating a maintainable system, that is designed for simplicity. The new architecture is modular, service oriented, and centralised. But simplicity alone can not ensure the maintainability of the new production system. We need good operability, meaning easy routine tasks, enabling operations team to focus on high-value activities.

A good service-level monitoring system is required in order for the operators to know how the system is performing currently and how it has changed (see subsection 7.1). In terms of the production system this means monitoring the state of tasks, master, task pool, and the object store. The progress of individual tasks should be monitored to prevent failing tasks from stalling whole task pipelines. Master should be monitored for availability since its failure (and especially failure to recover) would cause a system wide stall. Task pool and the database should also be monitored for availability; however, perhaps even more important is to monitor their performance in order to make quick adjustments in face of bottlenecks. A simple dashboard showing only the most critical signals should be set up to allow operators to confirm the correct operation of the system at a glance.

Logs should be aggregated for better operability (see subsection 7.2). We are especially interested in the master and task logs, because they are the main active parts of the new production system. But tasks are distributed across the system. If each task just logged to the local hard disk drive (HDD) of its executing worker, the logs would become distributed across the system, forcing operators to waste efforts hunting them down each time something bad happens. Therefore, logs should be aggregated to one central place for easier access.

Platform-level monitoring equipment should be installed to ensure that there are enough compute, storage, and network resources, and that they are working as

expected (see subsection 7.4). This should also prevent underlying hardware problems from manifesting as service-level problems, which could waste serious operator efforts.

Back-pressure should be build into master-worker communication to avoid cascading failures (see subsection 7.5). Cascading failures occur when failure of one component spreads to another due to lack of proper failure handling. We see two possible ways for a failure of a component to spread to another in the new production system. First, the unavailability of the task pool will cause workers to stall, which might result in pressure between the master and the workers. Second, the unavailability of the database will again cause workers to stall and similarly result in pressure between the master and the workers. Therefore, we should implement back-pressure into the master-worker communication to prevent failures from cascading to master. A simple solution would be to instruct workers to inform master whenever they are unable to reach task pool or database. Master can then back-off task assignments accordingly.

We should containerise the production system to increase operability (see subsection 7.6). A container packages the application with its dependencies allowing it to be run wherever the container technology is supported. Consequently, packaging the master, the task pool, and the database clients into containers enables operators to easily, deploy, migrate and update them. Even more important is to package each task into a container so that they can be run on any worker without needing to install their dependencies first. This also makes adding new nodes to the system easier since they only need to be installed with the chosen container technology in order to execute tasks.

A container orchestration platform should be installed for efficient container management (see subsection 7.6). These platforms enable operators to easily describe how they wish applications to be deployed and managed, while the platform handles the rest. Moreover, most container orchestration platforms actually provide all the maintainability improvements listed above (service-level and platform monitoring, logging, and back-pressure) plus many other features (see table 1).

9.3 Implementation

Instead of implementing the new production system from scratch, we decided to use ready-made components (where available) to save in development and maintenance

costs. We compared several products to find out which of them would best match the new design.

9.3.1 Kubernetes

As of March 2018 there are only four major container orchestration platforms that are on-premises ready. Out of these four container orchestration platforms Kubernetes was chosen as the basis for the new production system since it most closely matched our list of desired features (see Table 1). Some features were absolutely necessary, like support for high availability or for bind mounting, but most of the other features were chosen with operability in mind, e.g. logging, monitoring, debugging, and different deployment strategies.

We elaborate what is bind mounting and why it is necessary to our system. Bind mounting is the ability to access file system directly from the container. It is necessary since it enables the only strategy to smoothly transition from NFS to the new distributed object store. Initially, each task inputs and outputs to the old NFS cluster as before, which is done by mounting the NFS directories to each worker node and by bind mounting these directories from each task container. Afterwards, files should be migrated to distributed object store one by one with the respective changes to related tasks.

Kubernetes is the base on top of which we build the new production system. Its main responsibilities are to distribute triggered tasks evenly among the workers and to keep important services, like master and task pool, running indefinitely. All this is done through Kubernetes' interface which enables containerised jobs to be scheduled for execution. What is left for operators to do is to simply describe how they wish their services to be deployed and Kubernetes handles the rest. For our purposes, there are basically two kinds of deployments, services and one-off jobs. As was already mentioned, services are for long running jobs (e.g. the new master). Kubernetes continuously manages deployed services towards their desired state, which is configured into the deployment description, e.g. that there should be three replicas of some service. This could mean killing or restarting, depending on the needed change. One-off jobs are expected to finish and Kubernetes only guarantees them to be run at least once.

The idea is to deploy the master, the task pool, and the distributed object store clients as long running services to Kubernetes and have the master deploy tasks as

one-off jobs through Kubernetes interface. Consequently, master can concentrate on checking task triggers whereas Kubernetes handles the load balancing and ensures the tasks to be executed to completion. Since all tasks and services are handled by Kubernetes we can leverage its excellent monitoring and logging capabilities and enjoy uniform metrics on all parts of the system. It is important to note that we only plan to deploy the distributed object store clients to Kubernetes, not the object store itself, which should reside outside Kubernetes in its own cluster, like the NFS cluster.

9.3.2 Brigade

Brigade is an event-based scripting framework for Kubernetes. It enables operators to describe simple and complex task pipelines using JavaScript. These pipelines can be triggered based on times, message queue events, Hypertext Transfer Protocol (HTTP) requests, or any other events. Brigade also includes a dashboard, called Kashti, for monitoring tasks in real-time.

A Brigade script should define at least one event handler. Brigade triggers these event handlers when it receives a matching event from a gateway. Events are project specific, meaning that an event will only be triggered for the explicitly declared project. An event includes the following information: which project it belongs to, the name of the event, the entity that triggered the event, an optional script to execute, and an optional payload containing event data. When Brigade receives an event, it loads the referenced project and starts a new worker. The worker then executes the script using the triggered event as the entry point.

Scripts consist of event handlers, jobs, and tasks. An event handler associates an event with a function to process the event. When triggered, an event handler is explicitly given two pieces of information: an event record, that contains information about the event, and a context record, that provides some information about the project (e.g. environment variables or secrets). An event handler typically declares one or more jobs. A job is a unit of work that is associated with a container image. When a job is executed, the associated container image is pulled from an image repository, and is executed in the Kubernetes cluster. A job may also optionally declare tasks, which are executed inside the associated container. Finally the results of the container (if any) are returned to the event handler.

We plan to use Brigade as follows. We set up a custom gateway that checks for new

or changed data and creates events named after them. Initially this gateway can be a simple script that polls each data sources' output directory, but later, after we have migrated from NFS to a database, this gateway can be changed to a simple service with a hook to database events. Each of the current tasks is containerised and published to a repository that is accessible from the Kubernetes cluster. Lastly, Brigade is provided with a JavaScript file containing an event handler for each of the containerised tasks.

9.3.3 Riak CS

We chose Riak CS to be the object store of the new production system since it most closely fulfilled our list of desired features. We arrived at this decision after comparing three highly available, high-performance, on-premises capable, object stores: Riak CS, LeoFS, and Swift (see 2). The compared features were selected in accordance to the design of the new object store.

9.3.4 Image Registry

Since tasks in the new production system are containers, the task pool should be a container image registry. But unfortunately we could not find much architecture information on any of the popular image registries, and thus we were unable to compare them to find out which of them would best fulfil our plan for the new task pool. Therefore, we recommend taking the simplest approach, that is, deploying the task images to a private Docker Hub. To avoid the unnecessary latency of having to pull an image every time a task executes, the `imagePullPolicy` property of each task should be set to `ifNotPresent`, which causes the assigned worker to first check for a local image before pulling from the registry. However, if the latency still proves to be too large, we recommend deploying an on-premises private image registry.

10 Conclusions

We arrived at the following conclusions while writing this thesis:

1. Designing a distributed system is hard due to the numerous trade-offs one has to make in the process.
2. Distributed systems are complicated by nature and thus maintainability should be the number one goal in their design.
3. Container orchestration platforms provide a very powerful abstraction, which leads to a maintainability improvement unparalleled by any other available tool.
4. Single-leader replication is surprisingly versatile when accompanied with a well chosen sharding strategy.

10.1 Summary of Contributions

This thesis makes the following contributions to knowledge:

1. We designed a distributed production system that is maintainable, performant, available, reliable, and scalable.
2. We compared and selected ready-made software to build the production system we designed.
3. The research part of this thesis is a concise document of its own which presents the essentials of distributed systems in a clear manner.

10.2 Future Research

This thesis can be extended in the future in the following ways:

1. A production system with a better data locality: arriving data is replicated among a subset of workers and the related tasks are assigned to these nodes.
2. A guideline for designing distributed systems. We believe that such a document would be very valuable anyone dealing with distributed systems.
3. Compared object stores should be additionally benchmarked.

Abbreviations

***I*-confluence** invariant confluence. 49, 51, 65

CALM consistency as logical monotonicity. 49, 50, 65

CAP Brewer's theorem. 39, 41–46, 63

COPS Cluster of Order-Preserving Servers. 47, 48

CPU central processing unit. 4, 29, 31

CRDT commutative replicated data type. 49–51, 65, 68

FLP Fischer's, Lynch's, and Paterson's impossibility proof. 39–41, 63

FMI Finnish Meteorological Institute. i, 1, 59

GC garbage collector. 6

GFS Google File System. 24, 25, 57

GPS Global Positioning System. 18

GUI graphical user interface. 60

HDD hard disk drive. 69

HPC high performance computing. 4

HTTP Hypertext Transfer Protocol. 72

I/O input/output. 1, 60, 61

ID identifier. 21

LAN local area network. 4, 5, 40, 59

LWW last-write-wins. 1, 20, 69

MR Monotonic reads. 34, 35

MTBF mean time between failures. 58

MW Monotonic writes. 34, 35

NFS Network File System. 1, 59–62, 67, 71–73

NTP Network Time Protocol. 8

PACELC Abadi's extension to the CAP theorem. 39, 44, 45, 63

PODC The Symposium on Principles of Distributed Computing. 41

RAM random-access memory. 4

RMS root mean square. 8

RTC real-time clock. 17

RTT round-trip time. 29

RYW Read your writes. 34, 35

SI Snapshot Isolation. 30

SLA service-level agreement. 26, 46

SSI Serialisable Snapshot Isolation. 30, 31

TPS Toyota Production System. 3

WAN wide area network. 4, 5

WFR Writes follow reads. 34, 35

ZAB ZooKeeper Atomic Broadcast. 36

References

1 D. Klahr, P. Langley, and R. Neches, eds., *Production System Models of Learning and Development.* Production system models of learning and development., Cambridge, MA, US: The MIT Press, 1987.

2 T. Ohno, *Toyota Production System: Beyond Large-Scale Production.* CRC Press, Mar. 1988.

3 M. Bellgran and E. K. Säfsten, *Production Development: Design and Operation of Production Systems*. Springer Science & Business Media, Nov. 2009.

4 T. Steen, Marten van, Andrew S., *Distributed Systems*. Maarten van Steen, 3 ed., 2017.

5 W. L. Heimerdinger and C. B. Weinstock, "A conceptual framework for system fault tolerance," tech. rep., CARNEGIE-MELLON UNIV PITTSBURGH PA SOFTWARE ENGINEERING INST, 1992.

6 B. C. Neuman, "Scale in Distributed Systems," 1994.

7 M. Kleppmann, *Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems*. Beijing Boston Farnham Sebastopol Tokyo: O'Reilly Media, 1 edition ed., Apr. 2017.

8 L. A. Barroso, J. Clidaras, and U. Hölzle, "The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines, Second edition," *Synthesis Lectures on Computer Architecture*, vol. 8, no. 3, pp. 1–154, heinäkuu 31, 2013.

9 J. Hodges, "Notes on Distributed Systems for Young Bloods – Something Similar." https://www.somethingsimilar.com/2013/01/14/notes-on-distributed-systems-for-young-bloods/, 2013.

10 G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: Amazon's highly available key-value store," *ACM SIGOPS operating systems review*, vol. 41, no. 6, pp. 205–220, 2007.

11 S. Ghemawat, H. Gobioff, and S.-T. Leung, "The Google file system," in *ACM SIGOPS Operating Systems Review*, vol. 37, pp. 29–43, ACM, 2003.

12 A. Rotem-Gal-Oz, "Fallacies of distributed computing explained," *URL http://www. rgoarchitects. com/Files/fallacies. pdf*, vol. 20, 2006.

13 P. Bailis and K. Kingsbury, "The network is reliable," *Queue*, vol. 12, no. 7, p. 20, 2014.

14 W. Oremus and W. Oremus, "The Global Internet Is Being Attacked by Sharks, Google Confirms," *Slate*, Aug. 2014.

15 M. Imbriaco, "Downtime last Saturday." https://github.com/blog/1364-downtime-last-saturday, 2012.

16 M. A. Donges, "Re: Bnx2 cards intermittantly going offline — Netdev." https://www.spinics.net/lists/netdev/msg210485.html, 2012.

17 R. Govindan, I. Minei, M. Kallahalla, B. Koley, and A. Vahdat, "Evolve or Die: High-Availability Design Principles Drawn from Googles Network Infrastructure," pp. 58–72, ACM Press, 2016.

18 M. Caporaloni and R. Ambrosini, "How closely can a personal computer clock track the UTC timescale via the internet?," *European Journal of Physics*, vol. 23, pp. L17–L21, July 2002.

19 Y. Saito and M. Shapiro, "Optimistic replication," *ACM Computing Surveys (CSUR)*, vol. 37, no. 1, pp. 42–81, 2005.

20 J. Gray, P. Helland, P. O'Neil, and D. Shasha, "The dangers of replication and a solution," *ACM SIGMOD Record*, vol. 25, no. 2, pp. 173–182, 1996.

21 L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Communications of the ACM*, vol. 21, no. 7, pp. 558–565, 1978.

22 R. Schwarz and F. Mattern, "Detecting causal relationships in distributed computations: In search of the holy grail," *Distributed computing*, vol. 7, no. 3, pp. 149–174, 1994.

23 N. Preguiça, C. Baquero, P. S. Almeida, V. Fonte, and R. Gonçalves, "Dotted version vectors: Logical clocks for optimistic replication," *arXiv preprint arXiv:1011.5808*, 2010.

24 D. S. Parker, G. J. Popek, G. Rudisin, A. Stoughton, B. J. Walker, E. Walton, J. M. Chow, D. Edwards, S. Kiser, and C. Kline, "Detection of mutual inconsistency in distributed systems," *IEEE transactions on Software Engineering*, no. 3, pp. 240–247, 1983.

25 J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, and P. Hochschild, "Spanner: Google's globally distributed database," *ACM Transactions on Computer Systems (TOCS)*, vol. 31, no. 3, p. 8, 2013.

26 D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser, "Managing update conflicts in Bayou, a weakly connected replicated storage system," in *ACM SIGOPS Operating Systems Review*, vol. 29, pp. 172–182, ACM, 1995.

27 J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, and W. Weimer, "Oceanstore: An architecture for global-scale persistent storage," *ACM Sigplan Notices*, vol. 35, no. 11, pp. 190–201, 2000.

28 "Git - Documentation." https://git-scm.com/doc.

29 M. Burrows, "The Chubby lock service for loosely-coupled distributed systems," in *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, pp. 335–350, USENIX Association, 2006.

30 A. Fox and E. A. Brewer, "Harvest, yield, and scalable tolerant systems," in *Hot Topics in Operating Systems, 1999. Proceedings of the Seventh Workshop On*, pp. 174–178, IEEE, 1999.

31 F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable: A distributed storage system for structured data," *ACM Transactions on Computer Systems (TOCS)*, vol. 26, no. 2, p. 4, 2008.

32 W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen, "Don't settle for eventual: Scalable causal consistency for wide-area storage with COPS," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pp. 401–416, ACM, 2011.

33 B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni, "PNUTS: Yahoo!'s hosted data serving platform," *Proceedings of the VLDB Endowment*, vol. 1, no. 2, pp. 1277–1288, 2008.

34 D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin, "Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the World Wide Web," in *Proceedings of the Twenty-Ninth Annual ACM Symposium on Theory of Computing*, pp. 654–663, ACM, 1997.

35 M. P. Herlihy and J. M. Wing, "Linearizability: A correctness condition for concurrent objects," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 12, no. 3, pp. 463–492, 1990.

36 P. Bailis, "Linearizability versus Serializability | Peter Bailis." http://www.bailis.org/blog/linearizability-versus-serializability/, 2014.

37 M. Stonebraker, S. Madden, D. J. Abadi, S. Harizopoulos, N. Hachem, and P. Helland, "The end of an architectural era:(it's time for a complete rewrite)," in *Proceedings of the 33rd International Conference on Very Large Data Bases*, pp. 1150–1160, VLDB Endowment, 2007.

38 M. J. Cahill, U. Röhm, and A. D. Fekete, "Serializable isolation for snapshot databases," *ACM Transactions on Database Systems*, vol. 34, pp. 1–42, Dec. 2009.

39 H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil, "A critique of ANSI SQL isolation levels," *arXiv preprint cs/0701157*, 2007.

40 A. Adya and B. H. Liskov, *Weak Consistency: A Generalized Theory and Optimistic Implementations for Distributed Transactions*. PhD thesis, Massachusetts Institute of Technology, Dept. of Electrical Engineering and Computer Science, 1999.

41 L. Lamport, "How to make a multiprocessor computer that correctly executes multiprocess program," *IEEE transactions on computers*, no. 9, pp. 690–691, 1979.

42 H. Attiya and J. L. Welch, "Sequential consistency versus linearizability," *ACM Transactions on Computer Systems (TOCS)*, vol. 12, no. 2, pp. 91–122, 1994.

43 K. Birman and T. Joseph, *Exploiting Virtual Synchrony in Distributed Systems*, vol. 21. ACM, 1987.

44 M. Ahamad, G. Neiger, J. E. Burns, P. Kohli, and P. W. Hutto, "Causal memory: Definitions, implementation, and programming," *Distributed Computing*, vol. 9, no. 1, pp. 37–49, 1995.

45 P. Bailis, A. Fekete, A. Ghodsi, J. M. Hellerstein, and I. Stoica, "The potential dangers of causal consistency and an explicit solution," in *Proceedings of the Third ACM Symposium on Cloud Computing*, p. 22, ACM, 2012.

46 D. Terry, "Replicated data consistency explained through baseball," *Communications of the ACM*, vol. 56, no. 12, pp. 82–89, 2013.

47 D. B. Terry, A. J. Demers, K. Petersen, M. J. Spreitzer, M. M. Theimer, and B. B. Welch, "Session guarantees for weakly consistent replicated data," in *Parallel and Distributed Information Systems, 1994., Proceedings of the Third International Conference On*, pp. 140–149, IEEE, 1994.

48 M. J. Fischer, N. A. Lynch, and M. S. Paterson, "Impossibility of distributed consensus with one faulty process," *Journal of the ACM (JACM)*, vol. 32, no. 2, pp. 374–382, 1985.

49 C. Cachin, R. Guerraoui, and L. Rodrigues, *Introduction to Reliable and Secure Distributed Programming.* Berlin: Springer, 2nd ed. 2011 edition ed., Feb. 2011.

50 D. Ongaro and J. K. Ousterhout, "In search of an understandable consensus algorithm.," in *USENIX Annual Technical Conference*, pp. 305–319, 2014.

51 F. P. Junqueira, B. C. Reed, and M. Serafini, "Zab: High-performance broadcast for primary-backup systems," in *Dependable Systems & Networks (DSN), 2011 IEEE/IFIP 41st International Conference On*, pp. 245–256, IEEE, 2011.

52 X. Défago, A. Schiper, and P. Urbán, "Total order broadcast and multicast algorithms: Taxonomy and survey," *ACM Computing Surveys (CSUR)*, vol. 36, no. 4, pp. 372–421, 2004.

53 T. D. Chandra and S. Toueg, "Unreliable failure detectors for reliable distributed systems," *Journal of the ACM (JACM)*, vol. 43, no. 2, pp. 225–267, 1996.

54 L. Lamport, "Paxos made simple," *ACM Sigact News*, vol. 32, no. 4, pp. 18–25, 2001.

55 R. Guerraoui and A. Schiper, "Consensus: The big misunderstanding [distributed fault tolerant systems]," pp. 183–188, IEEE, 1997.

56 S. Gilbert and N. Lynch, "Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services," *Acm Sigact News*, vol. 33, no. 2, pp. 51–59, 2002.

57 S. Gilbert and N. Lynch, "Perspectives on the CAP Theorem," *Computer*, vol. 45, pp. 30–36, Feb. 2012.

58 M. Kleppmann, "A Critique of the CAP Theorem," *arXiv:1509.05393 [cs]*, Sept. 2015.

59 D. Abadi, "Consistency tradeoffs in modern distributed database system design: CAP is only part of the story," *Computer*, vol. 45, no. 2, pp. 37–42, 2012.

60 E. Brewer, "CAP twelve years later: How the "rules" have changed," *Computer*, vol. 45, pp. 23–29, Feb. 2012.

61 J. Brutlag, *Speed Matters for Google Web Search*. 2009.

62 A. Lakshman and P. Malik, "Cassandra: A decentralized structured storage system," *ACM SIGOPS Operating Systems Review*, vol. 44, p. 35, Apr. 2010.

63 "MongoDB for GIANT Ideas | MongoDB." https://www.mongodb.com/.

64 E. A. Brewer, "Lessons from giant-scale services," *IEEE Internet Computing*, vol. 5, no. 4, pp. 46–55, 2001.

65 M. Mavronicolas and D. Roth, "Linearizable read/write objects," *Theoretical Computer Science*, vol. 220, no. 1, pp. 267–319, 1999.

66 R. J. Lipton and J. S. Sandberg, *PRAM: A Scalable Shared Memory*. Princeton University, Department of Computer Science, 1988.

67 P. Mahajan, L. Alvisi, and M. Dahlin, "Consistency, availability, and convergence," *University of Texas at Austin Tech Report*, vol. 11, 2011.

68 H. Attiya, F. Ellen, and A. Morrison, "Limitations of Highly-Available Eventually-Consistent Data Stores," pp. 385–394, ACM Press, 2015.

69 P. Bailis, A. Fekete, M. J. Franklin, A. Ghodsi, J. M. Hellerstein, and I. Stoica, "Coordination avoidance in database systems," *Proceedings of the VLDB Endowment*, vol. 8, no. 3, pp. 185–196, 2014.

70 P. Alvaro, N. Conway, J. M. Hellerstein, and W. R. Marczak, "Consistency Analysis in Bloom: A CALM and Collected Approach.," in *CIDR*, pp. 249–260, 2011.

71 M. Letia, N. Preguiça, and M. Shapiro, "CRDTs: Consistency without concurrency control," *arXiv preprint arXiv:0907.0929*, 2009.

72 M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski, "Conflict-Free Replicated Data Types," in *Stabilization, Safety, and Security of Distributed Systems* (X. Défago, F. Petit, and V. Villain, eds.), vol. 6976, pp. 386–400, Berlin, Heidelberg: Springer Berlin Heidelberg, 2011.

73 P. Reynolds, J. L. Wiener, J. C. Mogul, M. K. Aguilera, and A. Vahdat, "WAP5: Black-box performance debugging for wide-area systems," in *Proceedings of the 15th International Conference on World Wide Web*, pp. 347–356, ACM, 2006.

74 B. H. Sigelman, L. A. Barroso, M. Burrows, P. Stephenson, M. Plakal, D. Beaver, S. Jaspan, and C. Shanbhag, "Dapper, a large-scale distributed systems tracing infrastructure," tech. rep., Technical report, Google, Inc, 2010.

75 P. Reynolds, C. E. Killian, J. L. Wiener, J. C. Mogul, M. A. Shah, and A. Vahdat, "Pip: Detecting the Unexpected in Distributed Systems.," in *NSDI*, vol. 6, pp. 9–9, 2006.

76 R. Fonseca, G. Porter, R. H. Katz, S. Shenker, and I. Stoica, "X-trace: A Pervasive Network Tracing Framework," in *Proceedings of the 4th USENIX Conference on Networked Systems Design & Implementation*, NSDI'07, (Berkeley, CA, USA), pp. 20–20, USENIX Association, 2007.

77 B. Burns and D. Oppenheimer, "Design Patterns for Container-based Distributed Systems.," in *HotCloud*, 2016.

78 C. Pahl, "Containerization and the paas cloud," *IEEE Cloud Computing*, vol. 2, no. 3, pp. 24–31, 2015.

79 W. Felter, A. Ferreira, R. Rajamony, and J. Rubio, "An updated performance comparison of virtual machines and linux containers," pp. 171–172, IEEE, 2015.

80 E. Casalicchio, "Autonomic Orchestration of Containers: Problem Definition and Research Challenges," ACM, 2017.

81 J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," in *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, (Berkeley, CA, USA), pp. 10–10, USENIX Association, 2004.

82 "ecFlow - ECMWF Confluence Wiki." https://software.ecmwf.int/wiki/display/ECFLOW/Doc

83 "Nagios Documentation." https://www.nagios.org/documentation/.

84 “Website performance and availability monitoring.” https://www.pingdom.com/.

85 “APM | New Relic Documentation.” https://docs.newrelic.com/docs/apm.

86 “Hashicorp/nomad: Nomad is a flexible, enterprise-grade cluster scheduler designed to easily integrate into existing workflows. Nomad can run a diverse workload of micro-service, batch, containerized and non-containerized applications. Nomad is easy to operate and scale and integrates seamlessly with Consul and Vault..” https://github.com/hashicorp/nomad.

87 “Documentation - Nomad by HashiCorp.” https://www.nomadproject.io/docs/index.html.

88 “Kubernetes/kubernetes: Production-Grade Container Scheduling and Management.” https://github.com/kubernetes/kubernetes.

89 “Setup | Kubernetes.” https://kubernetes.io/docs/setup/.

90 “Tasks | Kubernetes.” https://kubernetes.io/docs/tasks/.

91 “Concepts | Kubernetes.” https://kubernetes.io/docs/concepts/.

92 “Docker/swarmkit: A toolkit for orchestrating distributed systems at any scale. It includes primitives for node discovery, raft-based consensus, task scheduling and more..” https://github.com/docker/swarmkit.

93 “Docker Documentation | Docker Documentation.” https://docs.docker.com/.

94 “Dcos/dcos: DC/OS - The Datacenter Operating System.” https://github.com/dcos/dcos.

95 “Documentation for Mesosphere DC/OS 1.10 - Mesosphere DC/OS Documentation.” https://docs.mesosphere.com/1.10/.

96 “Leo-project/leofs: The LeoFS Storage System.” https://github.com/leo-project/leofs/.

97 “LeoFS Documentation.” https://leo-project.net/leofs/docs/index.html.

98 “Basho/riak_cs: Riak CS is simple, available cloud storage built on Riak..” https://github.com/basho/riak_cs.

99 “Riak Cloud Storage.” http://docs.basho.com/riak/cs/2.1.1/.

100 “Openstack/swift: OpenStack Storage (Swift).” https://github.com/openstack/swift.

101 “OpenStack Docs: Welcome to Swift’s documentation!.” https://docs.openstack.org/swift/latest/index.html.

Appendix 1. Container Orchestration Platform Comparison

Features	Nomad [86, 87]	Kubernetes [88, 89, 90, 91]	Docker Swarm Mode [92, 93]	DC/OS [94, 95]
License	Mozilla Public License 2.0	Apache License 2.0	Apache License 2.0	Apache License 2.0
Community size	3000+ stars, 200+ contributors	32000+ stars, 1500+ contributors	1500+ stars, 90+ contributors	1900+ stars, 100+ contributors
Container agnostic	Yes	Yes	No	Yes
High availability	Yes	Yes	Yes	Yes
Log storage	Logs are stored some hours	Logs are stored indefinitely	According to the selected driver	No
Log rotation	Yes	Requires third-party	According to the selected driver	No
Log access interface	Yes	Yes	Yes	Yes
Container-level metrics interface	Yes	Yes	Yes	Yes
Container-level metrics visualisation	Yes	Yes	Requires third-party	Yes
Platform-level metrics interface	Yes	Yes	Yes	Yes
Platform-level metrics visualisation	Yes	Yes	Yes	Yes
Debugging	No	No	No	No
Rolling updates	Yes	Yes	Yes	Yes
Roll-back	Yes	Yes	Yes	No
Blue-green deployments	Yes	No	No	No
Event-based job scheduler	No	Brigade	No	Scale
Bind mounts	No	Yes (hostPath)	Yes	No

Table 1: Comparison of some of the most popular container orchestration platforms.

Appendix 2. Distributed Object Storage Comparison

Features	LeoFS [96, 97]	Riak CS [98, 99]	Swift [100, 101]
License	Apache License 2.0	Apache License 2.0	Apache License 2.0
Community size	1100+ stars, 10+ contributors	500+ stars, 30+ contributors	1600+ stars, 200+ contributors
Multi-data centre	2	enterprise edition only	at least 2
Max object size	N/A	potentially multiple terabytes	virtually unlimited
Large objects are chunked	yes, configurable chunk size	yes, configurable chunk size	yes, configurable chunk size
Consistency model	eventual	eventual	eventual
Replication strategy Sloppy quorums	leaderless not yet	leaderless yes	leaderless no
Sharding scheme	consistent hashing	consistent hashing	consistent hashing
Shard size Convergence	one object per shard async read repair with background anti-entropy	one object per shard read-repair with background anti-entropy, both are highly configurable	one object per shard background anti-entropy (LWW)

Table 2: Distributed Object Storage Comparison.